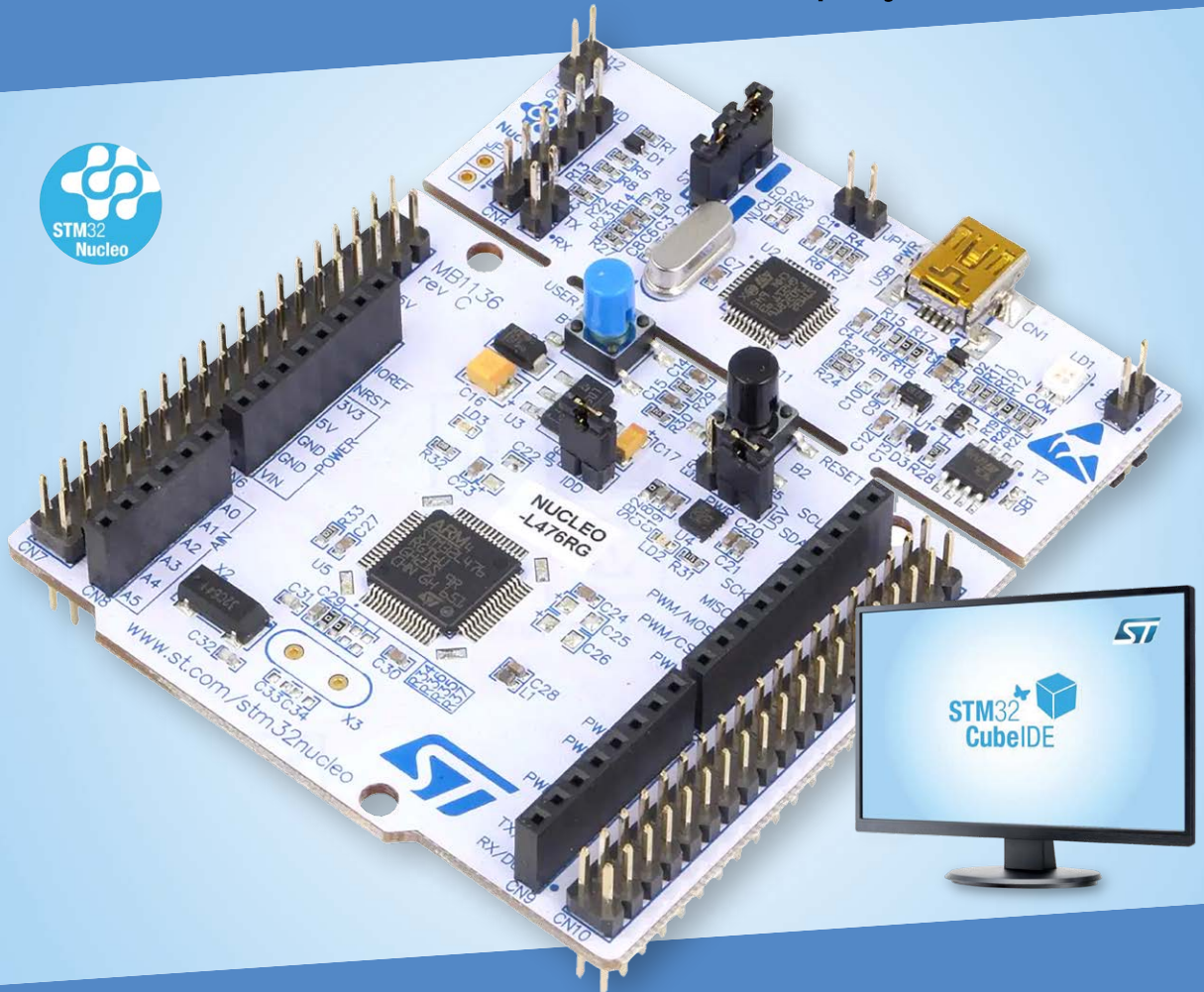


Nucleo Boards Programming with the STM32CubeIDE

Hands-on in more than 50 projects



Dogan Ibrahim

Nucleo Boards Programming with the STM32CubeIDE

Hands-on in more than 50 projects



Dogan Ibrahim

● This is an Elektor Publication. Elektor is the media brand of
Elektor International Media B.V.
PO Box 11, NL-6114-ZG Susteren, The Netherlands
Phone: +31 46 4389444

● All rights reserved. No part of this book may be reproduced in any material form, including photocopying, or storing in any medium by electronic means and whether or not transiently or incidentally to some other use of this publication, without the written permission of the copyright holder except in accordance with the provisions of the Copyright Designs and Patents Act 1988 or under the terms of a licence issued by the Copyright Licensing Agency Ltd., 90 Tottenham Court Road, London, England W1P 9HE. Applications for the copyright holder's permission to reproduce any part of the publication should be addressed to the publishers.

● **Acknowledgements**

The following figures and pictures in this book, numbered: 1.1, 1.5, 1.6, 1.7, 1.9, 2.2, 2.3, 2.9, 2.10 through 2.20, 2.22 through 2.32, 4.1, 4.2, 6.1, 14.1, 15.1 through 15.10, 16.2, and 16.7 are taken from these STMicroelectronics sources:

- UM1724 User Manual, STM32 Nucleo-64 Boards (DocID025833 Rev 14);
- RM0351 Reference Manual STM32L4x5 and STM32L4x6 advanced ARM®-based 32-bit MCUs (DocID024597 Rev 5).

The pictures of Nucleo Expansion Boards in Chapter 16 of the book are taken from the following STMicroelectronics Internet source:

<http://www.st.com/en/evaluation-tools/stm32-nucleo-expansion-boards.html?querycriteria=productId=SC1971>.

All the above figures/pictures are used with the written permission of:

© STMicroelectronics. Used with permission.

The author would like to thank to Michael Markowitz of the STMicroelectronics for giving permission to reproduce the above pictures/figures, as well as for providing sample Nucleo expansion boards for use in the projects contained in the book. The author is also grateful to Elektor International Media for editing and publishing the book.

● The author and publisher have used their best efforts in ensuring the correctness of the information contained in this book. They do not assume, and hereby disclaim, any liability to any party for any loss or damage caused by errors or omissions in this book, whether such errors or omissions result from negligence, accident or any other caus

● British Library Cataloguing in Publication Data

A catalogue record for this book is available from the British Library

● **ISBN 978-3-89576-416-5** Print

ISBN 978-3-89576-416-6 eBook

ISBN 978-3-89576-416-7 ePub

● © Copyright 2020: Elektor International Media B.V.
Prepress Production: D-Vision, Julian van den Berg

Elektor is part of EIM, the world's leading source of essential technical information and electronics products for pro engineers, electronics designers, and the companies seeking to engage them. Each day, our international team develops and delivers high-quality content - via a variety of media channels (including magazines, video, digital media, and social media) in several languages - relating to electronics design and DIY electronics. **www.elektormagazine.com**

PREFACE	11
CHAPTER 1 • STM32 Nucleo Development Boards	12
1.1 Overview	12
1.2 STM32 Nucleo development boards	12
1.2.1 STM32 processor family numbering	13
1.2.2 Nucleo-32 development boards	15
1.2.3 Nucleo-64 development boards	16
1.2.4 Nucleo-144 development boards	17
1.3 The Nucleo-L476RG development board	18
1.3.1 Two-part board	18
1.3.2 The power supply	21
1.3.3 The LEDs	22
1.3.4 Pushbutton switches	22
1.3.5 Jumper JP6	22
1.3.6 The ST-LINK/V2-1	22
1.3.7 Input-Output connectors	23
1.3.8 The demo software	24
1.4 Summary	25
CHAPTER 2 • STM32 Nucleo Processor Architecture	26
2.1 Overview	26
2.2 Arm processors	26
2.2.1 Cortex-M	28
2.2.2 Cortex-R	28
2.2.3 Cortex-A	29
2.2.4 Cortex-M processor comparison	29
2.2.5 Processor performance measurement	29
2.2.6 Cortex-M compatibility	30
2.2.7 Choice of an STM32 processor	30
2.3 The STM32L476RGT6 microcontroller	30
2.3.1 Basic features of the STM32L476RGT6	31
2.3.2 Internal block diagram	31
2.3.3 General purpose inputs and outputs (GPIOs)	34

2.3.4 Electrical characteristics	42
2.3.5 The power supply	43
2.3.6 Low power modes	44
2.3.7 The clock circuit.	45
2.3.8 Analogue to digital converter (ADC)	48
2.3.9 Digital to analogue converter (DAC)	48
2.3.10 Timers	49
2.3.11 Interrupts	49
2.4 Summary	55
CHAPTER 3 • STM32 Nucleo Software Development Tools (Toolchains)	56
3.1 Overview	56
3.2 Integrated development environments supporting the Nucleo boards.	56
3.3 Embedded Workbench for Arm (EWARM)	56
3.3.1 Installing the EWARM	57
3.4 Arm Mbed.	58
3.5 MDK-ARM	60
3.6 TrueSTUDIO	61
3.7 System Workbench for STM32 (SW4STM32)	62
3.8 STM32CubeIDE	64
3.9 Summary	66
CHAPTER 4 • Example Project — Using the Mbed	67
4.1 Overview	67
4.2 Using the ARM Mbed	67
4.3 Summary	71
CHAPTER 5 • STM32CubeIDE Nucleo-L476 Projects	72
5.1 Overview	72
5.1.1 STM32cubeIDE GPIO library	72
5.2 Project 1: Lighthouse flashing LED.	75
5.3 Project 2: Alternately Flashing LEDs.	89
5.4 Project 3: ‘Moving’ LEDs.	95
5.5 Project 4: Binary Up Counter with LEDs	101
5.6 Project 5: Random Flashing LEDs	106

5.7 Project 6: Pushbutton and LED	110
5.8 Project 7: Control of Multiple LEDs by 2 Buttons	115
5.9 Project 8: LED Dice	123
5.10 Project 9: 7-Segment LED Counter	132
5.11 Project 10: Two-Digit Multiplexed 7-Segment LED	140
5.12 Project 11: External interrupt to control an LED	148
5.13 Project 12: Two-digit Interrupt-Driven 7-Segment Event Counter	157
5.14 Project 13: Four-Digit 7-Segment LED Display.	163
5.15 Project 14: Interrupt-Based Up/Down Counter with Four-Digit 7-Segment LED Display	170
5.16 Project 15: Multiple External Interrupts Sharing the Same Interrupt Line	180
5.17 Summary	186
CHAPTER 6 • Timers	187
6.1 Overview	187
6.2 STM32 timers	187
6.3 Setting a timer	189
6.4 Project 1: Timer Interrupt to Flash LED Every Second	190
6.5 Project 2: 4-Digit 7-Segment LED Up Counter with Timer Interrupts	195
6.6 Summary	204
CHAPTER 7 • LCD Displays	205
7.1 Overview	205
7.2 Project 1: Using parallel LCDs – Displaying Text	205
7.3 Project 2: Using LCDs – Simple Up Counter	221
7.4 Summary	225
CHAPTER 8 • Using the Analogue to Digital Converters	226
8.1 Overview	226
8.2 The STM32 ADC conversion modes	226
8.3 Project 1: Analogue Voltmeter (polling ADC).	228
8.4 Project 2: ADC with Multiple Inputs (polling ADC)	237
8.5 Project 3: Single-input ADC with Conversion Interrupt.	246
8.6 Project 4: Analogue Temperature Sensor	251
8.7 Project 5: ON-OFF Temperature Controller	258

8.8 Project 6: Multiple-input ADC with DMA	266
8.9 Timer-driven ADC	276
8.10 External-driven ADC.	276
8.11 ADC calibration	276
8.12 Summary	276
CHAPTER 9 • Using the Digital-to-Analogue Converters	277
9.1 Overview	277
9.2 Project 1: Sawtooth Waveform Generator with Manual DAC Driving.	277
9.3 Project 2: Squarewave Generator with Manual DAC Driving	285
9.4 Project 3: Sinewave Generator with Manual DAC Driving	286
9.5 Project 4: Arbitrary Waveform Generator with Manual DAC Driving	287
9.6 Project 5: Arbitrary Waveform Generator with timer-based DMA	289
9.7 Hardware waveform generation.	296
9.8 Project 6: Hardware-based Triangular Waveform Generation.	298
9.9 Noise signal generation	303
9.10 Summary	304
CHAPTER 10 • Pulsewidth Modulation (PWM)	305
10.1 Overview	305
10.2 Basic theory of pulsewidth modulation	305
10.3 Operation of the PWM	306
10.4 Project 1: Mosquito Repeller	308
10.5 Project 2: Continuously Variable Duty Cycle	316
10.6 Project 3: Multiple PWM Waveforms	318
10.7 Project 4: Potentiometer-controlled Duty Cycle Control of PWM Waveform.	325
10.8 Summary	333
CHAPTER 11 • Serial Communication.	334
11.1 Overview	334
11.2 UART ports of the Nucleo-L476RG development board	336
11.3 Serial communication program on a PC	337
11.4 Project 1: Displaying Text on the PC.	340
11.5 Project 2: Simple Up Counter	345
11.6 Project 3: Times Table	351

11.7 Project 4: Practising Elementary Multiplication	358
11.8 Project 5: Displaying Ambient Temperature on the PC Screen	362
11.9 Project 6: Communicating with Arduino (Displaying Temperature)	369
11.10 UART in interrupt mode	376
11.11 Project 7: Communicating with Arduino – UART Interrupt Mode.	376
11.12 Using UART in DMA mode	382
11.13 Summary	382
CHAPTER 12 • The I²C Bus Interface	383
12.1 Overview	383
12.2 The I ² C Bus.	383
12.3 STM32L476RG I ² C ports	384
12.4 Project 1: Port Expander.	385
12.5 Project 2: EEPROM memory	395
12.6 Project 3: TMP102 Temperature Sensor Chip Reading	405
12.7 Summary	414
CHAPTER 13 • SPI Bus Projects	415
13.1 Overview	415
13.2 Nucleo-L476RG SPI pins	416
13.3 Project 1: Port Expander.	417
13.4 Summary	427
CHAPTER 14 • Program Debugging	428
14.1 Overview	428
14.2 Project 1: Simple Debug.	428
14.3 Project 2: Debugging the GPIO	432
14.4 Project 3: Displaying Characters in Debug Window	433
14.5 Project 4: Using ‘printf’ to Display Data in Debug Window	436
14.6 Project 5: Using the ST-Link Virtual COM Port	438
14.7 Summary	438
CHAPTER 15 • STM32L4 MCU Power Management	439
15.1 Overview	439
15.2 Low power modes	439
15.3 Power modes transitions.	444

15.4 Low power peripherals	445
15.5 Debugging in low-power modes.	445
15.6 Measuring Nucleo current consumption	445
15.7 Project 1: Sleep Mode Example	445
15.8 Project 2: Stop Mode Example.	449
15.9 Project 3: Standby Mode Example	451
15.10 Summary	452
CHAPTER 16 • Using the Expansion Boards	453
16.1 Overview	453
16.2 Industrial Digital Output Expansion Board (X-NUCLEO-OUT01A1)	453
16.3 Project 1: Flashing an LED	456
16.4 Brushed DC Motor Driver Expansion Board (X-NUCLEO-IHM13A1)	459
16.5 Motion MEMS and Environmental Sensor Expansion Board (X-NUCLEO-IKS01A2)	461
16.6 Project 2: Reading the Temperature from the X-NUCLEO-IKS01A2 Expansion Board	464
16.7 Project 3: Using the X-CUBE-MEMS1 Library	478
16.8 Wi-Fi Expansion Board (X-NUCLEO-IDW01M1).	481
APPENDIX • FreeRTOS For the STM32 MCU	484
A.1 Overview	484
A.2 Multitasking kernel advantages	484
A.3 The need for an RTOS	485
A.4 The FreeRTOS	485
A.5 FreeRTOS project with the STM32MCubeIDE.	486
Index	494

PREFACE

Arm (previously written as ARM; ARM) is a family of processors based on the RISC architecture that requires fewer transistors than conventional processors. As a result, Arm processors cost less, have reduced power consumptions, and are therefore desirable for use in portable battery-operated devices. Currently, a very large percentage of smartphones, laptops, tablet computers, and portable games devices are all designed using the Arm architecture. Most Arm processors have 32-bit instruction sets with floating point support. It is estimated that over 100 billion Arm processors were produced in the year 2017.

Arm-Cortex is a family of 32-bit RISC processors consisting of Cortex-M0/M0+/M1/M3 and Cortex-M4. Lately, new models such as Cortex-M323 and Cortex-M33 are available. Although 8-bit microcontrollers have been very popular in the past, the devices in the Cortex-M series have become widespread replacements.

The STM32 Nucleo family of processors is manufactured by STMicroelectronics. These are low-cost Arm microcontroller development boards. This book is about developing projects using the Nucleo-L476RG development board with the STM32CubeIDE software. In the early chapters of the book the architecture of the Nucleo family is briefly described.

All the projects in the book have been designed using the STM32CubeIDE software development tool which can be downloaded free of charge from the STMicroelectronics website. The book covers many projects using most features of the Nucleo-L476RG development boards where the full software listing for STM32CubeIDE are given for every project with the description of each software. The projects range from simple flashing LEDs to more complex projects using modules and devices such as GPIO, ADC, DAC, I²C, SPI, LCD, power management, analogue inputs, and others. In addition, several projects are given using the compatible Nucleo Expansion Boards. These Expansion Boards plug on top of the Nucleo development boards and provide features such as industrial input/output, DC motor drive, stepper motor drive, environmental MEMS sensors, accelerometer, gyroscope, Wi-Fi, and many more.

All the projects in the book have been fully tested and are working. The following sub-headings are given for each project where possible:

- Project Title
- Description
- Aim
- Block Diagram
- Circuit Diagram
- Program Listing (based on STM32CubeIDE)

Complete program listings for each project can be obtained from the book support and resources page on the Elektor website, www.elektor.com/xxxxxxx.

I hope you find the projects interesting and the book becomes a useful source of reference for your future STM32 Nucleo projects based on the STM32CubeIDE.

Prof Dr Dogan Ibrahim

London, 2020

CHAPTER 1 • STM32 Nucleo Development Boards

1.1 Overview

This Chapter is about the STM32 Nucleo development boards, and brief specifications of their various guises are described here. In addition, the popular Nucleo-L476RG board used in the projects in this book is described in greater detail.

1.2 STM32 Nucleo development boards

STMicroelectronics is a company with a reputation of supplying quality microcontroller development boards and assorted hardware and software development tools. The Nucleo family of low-cost development boards comprises small but powerful boards based on the 32-bit Arm (also: 'ARM') Cortex architecture. These development boards are targeted for a large audience, including students, professional engineers, and hobbyists at all levels. The boards are compatible with the popular Arduino, mbed, ST-LINK, and ST Morpho, making them accessible to users with different backgrounds.

There are over 30 different boards in the Nucleo family, aimed to satisfy the needs of almost all types of user. The Nucleo boards come in three different sizes: small (**Nucleo-32**), short (**Nucleo-64**), and long (**Nucleo-144**) where the numbers refer to the pin counts of the MCUs. These three groups are further divided into three sub-groups: ultra-low power (**green**), mainstream (**blue**), and high performance (**magenta**).

The ultra-low power boards are based on the STM32 L family and these boards are targeted for low-power applications, such as watches, smart meters etc. Examples of the ultra-low power boards are: Nucleo-L011K4, Nucleo-L031K6 and Nucleo-L432KC. There are three sub-categories in the STM32 L family:

- L0, Arm Cortex-M0+
- L1, Arm Cortex-M3
- L4, Arm Cortex-M4

About half of the STM32 Nucleo boards are in the mainstream category. Examples of the mainstream boards are: Nucleo-F303K8, Nucleo-F042K6, Nucleo-F303RE etc. There are three sub-categories in the mainstream category:

- F0, Arm Cortex-M0+
- F1, Arm Cortex-M3
- F3, Arm Cortex-M4

The high-performance boards have large memories and faster MCUs. Examples of high-performance boards are: Nucleo-F410RB, Nucleo-F401RE, Nucleo-F722ZE, etc. There are three subcategories in the high-performance category:

- F2, Arm Cortex-M3
- F4, Arm Cortex-M4
- F7, Arm Cortex-M7

The Nucleo-32 boards are small (50 mm × 19 mm) and Arduino Nano compatible. The Nucleo-64 and Nucleo-144 are Arduino Uno compatible and they also have ST Morpho extension connectors which carry the MCU pins. There are large number of Arduino Nano/Uno compatible shields available in the market and these shields can easily be used with the

The STM32 processors are numbered as follows:

STM32xxyyzpqr

Where

xx is the processor code as shown in Table 1;

yy is a number;

z is the MCU pin number count as shown in Table 1.2.

Code (z)	Pin count
A	169
B	208
C	48
F	20
G	28
H	40
I	176
J	72
K	32
M	81
N	216
Q	132
R	64
T	36
U	63
V	100
Z	144

Table 1.2: MCU pin counts.

p is the size of the flash memory as shown in Table 1.3;

Code (p)	Flash memory (KB)
4	16
6	32
8	64
B	128
Z	192
C	256
D	384
E	512
F	768

G	1024
H	1536
I	2048

Table 1.3: Flash memory size.

q is the processor's case type as shown in Table 1.4;

Code (q)	Casing
T	LQFP
H	UFBGA
Y	WLCSP

Table 1.4: Processor casing.

r is the operating temperature range as shown in Table 1.5.

Code (r)	Temperature range
6	–40 to 85 °C
7	–40 to 105 °C

Table 1.5: Temperature range.

An example of a processor type number is given below.

STM32L031K6T6: **L0** family, number **31**, **32** pin device, **32 KB** flash memory, **LQFP** type packaging, **–40 to 85 °C** operating temperature range.

1.2.2 Nucleo-32 development boards

Figure 1.2 shows an example Nucleo-32 board, the Nucleo-L031K6. This is an ultra-low power low-cost board incorporating the STM32L031K6T6 microcontroller. The board is Arduino Nano compatible allowing many Arduino shields to be used with the board. This board has the following features:

- 32 MHz Cortex M0+ microcontroller in 32-pin package
- 32 KB flash memory
- 8 KB RAM
- 1 KB EEPROM
- Real-time clock
- Serial interfaces (USART, SPI, and I²C)
- Three LEDs (USB communication, power, user)
- Pushbutton Reset
- Flexible power supply options: ST-LINK USB VBUS or external sources
- Arduino Nano compatible expansion connector
- ST-LINK/V2-1 debugger/programmer with mass storage, virtual COM port, and debug port

- Support for Integrated development Environment software (IAR, Keil, Arm Mbed, GCC-based IDEs).

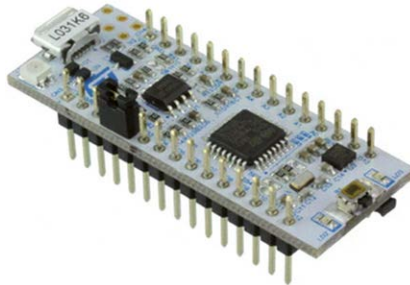


Figure 1.2: Nucleo-32 development board (Nucleo-L031K6).

1.2.3 Nucleo-64 development boards

Figure 1.3 shows an example Nucleo-64 board, the Nucleo-F091RC. This is a mainstream board incorporating a 64-pin MCU. The board is Arduino Uno compatible and as such, a large array of Arduino shields can be used with the board. This board has the following features:

- One user LED
- One user pushbutton switch
- 32.768 kHz crystal oscillator
- ST morpho connector
- Arduino Uno expansion socket
- Flexible power-supply options: ST-LINK USB V_{BUS} or external sources
- ST-LINK/V2-1 debugger/programmer with mass storage, virtual COM port, and debug port
- Comprehensive free software libraries
- Support of a wide choice of Integrated Development Environments (IAR, Keil, Arm Mbed, GCC-based IDEs)

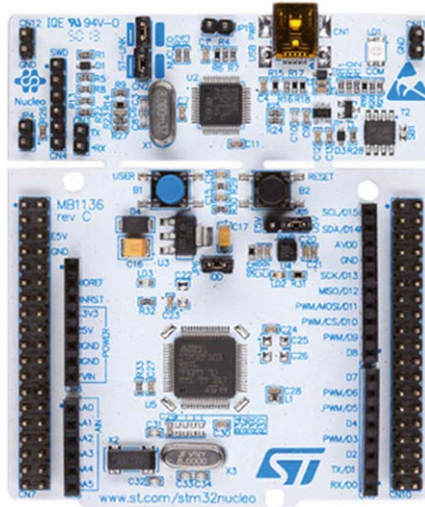


Figure 1.3: Nucleo-64 development board (Nucleo-F091RC).

1.2.4 Nucleo-144 development boards

Figure 1.4 shows an example Nucleo-144 board, the Nucleo-F722ZE. This is a high-performance board incorporating a 144-pin MCU. The board is Arduino Uno compatible. The features of this board are:

- Ethernet compliant with RJ45 connector
- ST morpho connector
- ST-LINK/V2-1 debugger/programmer with mass storage, virtual COM port, and debug port
- ST Zio connector
- 3 user LEDs
- 2 pushbutton switches
- 32.768 kHz crystal oscillator
- Flexible power-supply options: ST-LINK USB V_{BUS} or external sources
- Comprehensive free software libraries
- Support of a wide choice of Integrated Development Environments (IAR, Keil, Arm Mbed, GCC-based IDEs)



Figure 1.4: Nucleo-144 development board (Nucleo-F7222ZE).

1.3 The Nucleo-L476RG development board

The Nucleo-L476RG board is one of the most popular Nucleo development boards. This is a 64-pin ultra-low power board (see Figure 1.1). This board is used as the example development board in all the projects in this book. Therefore, full details of this board are given in this section so that the readers are familiar with this board. Further details about the Nucleo-64 boards can be obtained from the *STMicroelectronics User Manual UM1724, STM32 Nucleo-64 boards, 2017*.

1.3.1 Two-part board

The Nucleo-L476RG board measures 70 mm × 82.5 mm. As shown in Figure 1.5, the board consists of two parts: the smaller ST-LINK part with the mini USB port, and the MCU part. The ST-LINK part of the PCB can be cut if desired to reduce the overall board size. If this is done the MCU part can only be powered by V_{IN} , E5V and 3.3 V through the VIN on CN7 ST morpho connector, or 3.3 V on CN6 Arduino connector. It is possible to program the MCU after the ST-LINK part is cut by connecting wires between CN4 on the ST-LINK board and SWD signals on connector CN7 (pin 15, SWCLK; and pin 13, SWDIO).

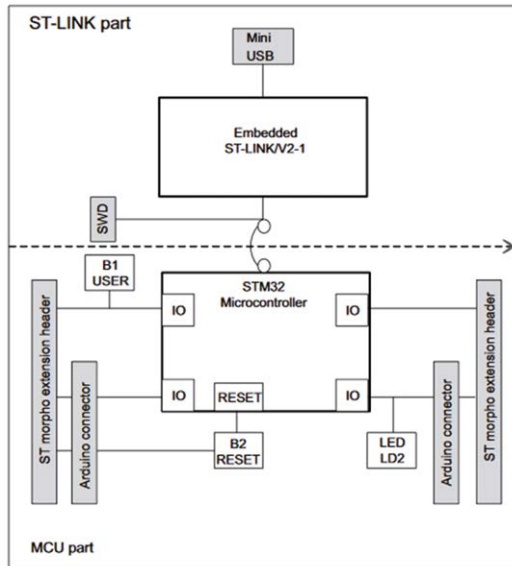


Figure 1.5: Nucleo-L476RG board parts. (©STMicroelectronics. Used with permission).

The components on the top side of the board are shown in Figure 1.6. Some of the important components that you may need to know their locations on the board are:

- CN1: mini USB socket
- CN2: ST-LINK/Nucleo selector
- B1: User pushbutton
- B2: Reset button
- LD1: red/green communications LED
- LD2: green user LED
- LD3: red power LED
- Arduino connectors
- ST morpho connectors

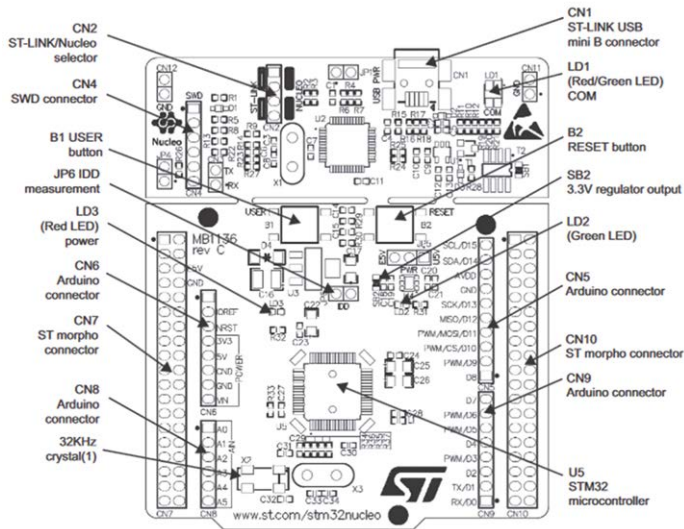


Figure 1.6: Components on the top side of the board.
(©STMicroelectronics. Used with permission).]

The bottom side of board is shown in Figure 1.7.

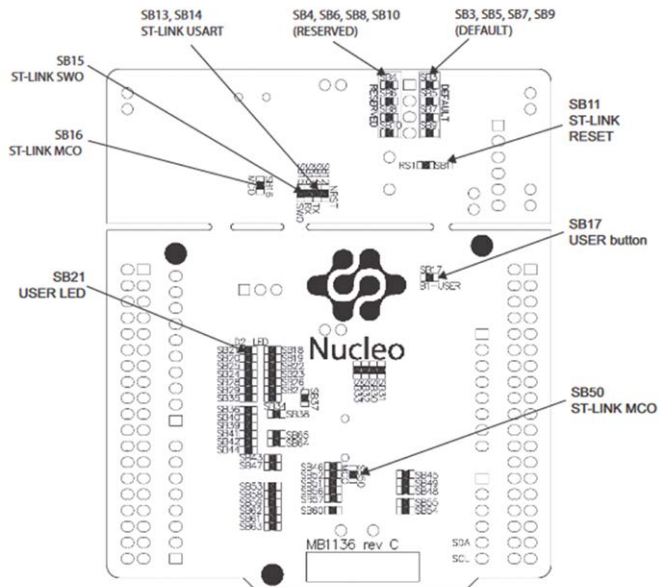


Figure 1.7: Bottom side of the board. (©STMicroelectronics. Used with permission).

1.3.2 The power supply

Power to the board is normally supplied through the mini USB connector CN1 using a USB cable, connected to a PC. It is also possible to use an external power source V_{IN} (7 V to 12 V), E5V (5 V) or +3.3 V.

Powering through the mini USB connector

When powered through the CN1 mini USB port (U5V), a jumper must be connected between pin 1 and pin 2 of jumper JP5 on the MCU board as shown in Figure 1.8. Notice that this is the default state of this jumper.

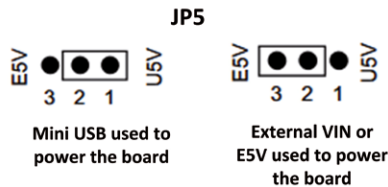


Figure 1.8: Powering the board through the mini USB port.

Jumper JP1 on the ST-LINK board can be configured depending on the maximum current consumption of the MCU board. This jumper should be OFF (default state) for an allowed current of 300 mA, and it should be ON for an allowable current of 100 mA. If the total current consumption of the board exceeds 300 mA then it is important that an external power supply must be used through VIN or E5V.

Powering through external inputs

External power can be applied through VIN or E5V inputs. When power is to be supplied by an external power supply, the following jumpers must be configured:

- jumper JP5 pin 2 and pin 3 connected;
- jumper JP1 OFF (removed).

V_{IN} can be supplied through pin 8 of connector CN6 or pin 24 of connector CN7. The voltage range must be between 7 VDC to 12 VDC. The maximum current capacity should be 800 mA when the input voltage is 7 V, 450 mA when the voltage is between 7 V and 9 V, and 250 mA when the voltage is greater than 9 V and less than 12 V.

E5V can be supplied through pin 6 of connector CN7. The voltage range is 4.75 V to 5.25 V, with maximum current capacity of 500 mA.

When power is supplied through V_{IN} or E5V, it is still possible to use the USB connector for communication, programming or debugging. In this situation, the board must first be powered using VIN or E5V and then the USB cable should be connected to the PC.

The following procedure should be followed if external power will be used and at the same time the USB connector will be used for communication/programming/debugging:

- connect pin 2 and pin 3 of jumper JP5;
- remove jumper JP1;
- connect external power V_{IN} or E5V;
- make sure that LD3 is turned ON;
- connect the USB connector to your PC.

External power input +3.3 V

It is also possible to power the board using external +3.3 V power supply. The range of this supply must be between +3 V to +3.6 V. When powered through the +3.3 V, the ST-LINK board is not powered and therefore the programming and debugging features are not available. This is usually the case when the ST-LINK part of the PCB has been cut. The +3.3 V external power supply must be applied to pin 4 of connector CN6.

Notice that when powered by USB, VIN or E5V, +5 V power is available at pin 5 of CN6 or pin 18 of CN7. These pins for example can be used to provide power to an extension board (e.g. to an Arduino).

1.3.3 The LEDs

LD1 is a tricolour LED (green, orange, red) providing visual indication about the ST-LINK communication. The default colour of this LED is red and it turns green when communication is in progress between the PC and the ST-LINK. The various states of LD1 are:

- slow blinking Red: at power-ON before USB initialization;
- fast blinking Red: after first communication between PC and ST-LINK/V2;
- red ON: initialization between PC and ST-LINK/V2 is complete;
- green ON: after successful target communication initialization;
- blinking Red/Green: during communication with target;
- green ON: communication finished successfully;
- orange ON: communication failure.

LD2 is a green colour user-controlled LED. This LED is connected to pin 11 of connector CN10 (STM32 I/O pin PA_5) on the Nucleo-L476RG board. The Arduino D13 port pin is also connected to LD2. Applying logic 1 turns ON the LED.

LD3 is the red power LED that indicates when +5 V power is available on the MCU board.

1.3.4 Pushbutton switches

There are two pushbutton switches on the MCU board called B1 and B2. B1 is the user button and is connected to STM32 I/O pin PC_13 (pin 23 on connector CN7 on the Nucleo-L476RG board). B2 is the Reset button used to reset the MCU.

1.3.5 Jumper JP6

This jumper is labelled IDD and can be used to measure the current consumption of the MCU by removing the jumper and then connecting an ammeter to the jumper pins. This jumper is ON by default.

1.3.6 The ST-LINK/V2-1

The ST-LINKV2-1 programming and debugging tool is integrated in the Nucleo boards and it makes the boards Mbed Enabled. ST-LINK/V2-1 supports only SWD for STM32 devices. The ST-LINK/V2-1 does not support SWIM interface and the minimum supported application voltage is limited to 3 V. The ST-LINK/V2-1 supports virtual COM port interface on USB, USB software reenumeration, mass storage interface on USB, and USB power management request for more than 100 mA power on USB.

There are two different ways to use the embedded ST-LINK/V2-1 depending on the CN2 jumper settings. When both CN2 jumpers are ON (the default state) then the ST-LINK/

V2-1 functions are enabled for on-board programming. When both CN2 jumpers are OFF then the ST-LINK/V2-1 functions are enabled for external CN4 connector (SWD supported). Before connecting the Nucleo-64 board to a Windows PC, a driver for ST-LINK/V2-1 must be installed. This can be downloaded from the following site. You will have to register at the site so that you can download the driver. At the time of writing this book the driver was called **en.stsw-link009.zip** and found here:

<http://www.st.com/en/development-tools/stsw-link009.html#getsoftware-scroll>

1.3.7 Input-Output connectors

Figure 1.9 shows the input-output connectors for the Nucleo-L476RG board. Notice that different models may have different pin configurations and you should check the appropriate User Manual for the correct configuration of the model you are using. CN5, CN6, CN8 and CN9 are Arduino Uno compatible connectors. CN7 and CN10 are the ST morpho connectors that carry most of the MCU signals.

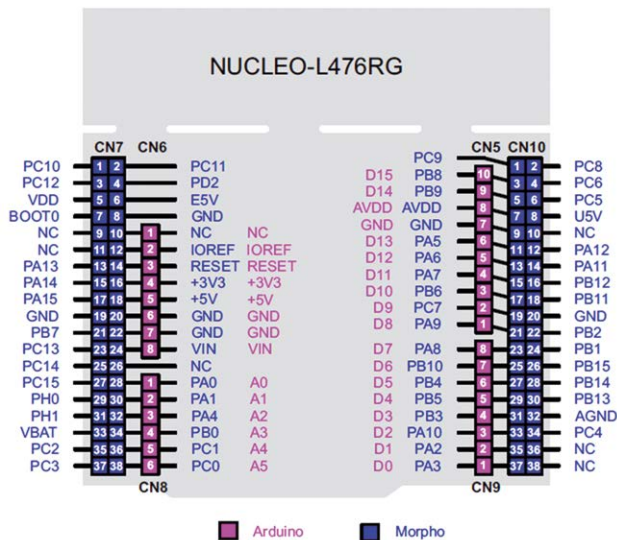


Figure 1.9: Nucleo-L476RG I/O connectors.
(©STMicroelectronics. Used with permission).

The Arduino Uno connector pin configuration on the Nucleo-476RG board is shown in Table 1.6.

Connector	Pin	Pin name	STM32 pin	Function
CN6 POWER	1	NC	-	-
	2	IOREF	-	3.3V Ref
	3	RESET	NRST	RESET
	4	+3.3V	-	3.3 V input/output
	5	+5V	-	5 V output
	6	GND	-	Ground
	7	GND	-	Ground
	8	VIN	-	Power input
CN8 ANALOG	1	A0	PA0	ADC12_IN5
	2	A1	PA1	ADC12_IN6
	3	A2	PA4	ADC12_IN9
	4	A3	PB0	ADC12_IN15
	5	A4	PC1	ADC123_IN2
	6	A5	PC0	ADC123_IN1

RIGHT CONNECTORS

CN5 DIGITAL	10	D15	PB8	I2C1_SCL
	9	D14	PB9	I2C1_SDA
	8	AREF	-	AVDD
	7	GND	-	GND
	6	D13	PA5	SPI1_SCK
	5	D12	PA6	SPI1_MISO
	4	D11	PA7	TIM17_CH1 or SPI1_MOSI
	3	D10	PB6	TIM4_CH1 or SPI1_CS
	2	D9	PC7	TIM3_CH2
	1	D8	PA9	-
CN9 DIGITAL	8	D7	PA8	-
	7	D6	PB10	TIM2_CH3
	6	D5	PB4	TIM3_CH1
	5	D4	PB5	-
	4	D3	PB3	TIM2_CH2
	3	D2	PA10	-
	2	D1	PA2	USART2_TX
	1	D0	PA3	USART2_RX

Table 1.6: Arduino UNO connector pin configuration.

1.3.8 The demo software

The Nucleo boards are shipped with a pre-loaded demo software. To run the demo software on the Nucleo-L476RG board, follow the sequence below:

- check that jumper JP1 is OFF, JP5 is ON, JP6 is ON, and CN2 are ON;
- connect the Nucleo-L476RG board connector CN1 to a PC with a mini USB cable (Type A to mini-B cable). The red LED LD3 (PWR) and LD1 (COM) should light up. LD1 (COM) and green LED LD2 should blink;
- press button B1 (left button);
- click button B1 and you should see the blinking rate of the green LED LD2 change accordingly.

If the demonstration software is run successfully then the board should be ready for use in projects.

1.4 Summary

In this Chapter we have learned about the types of STM Nucleo development boards. In addition, the specifications and configuration of the popular Nucleo-L476RG board have been described in greater detail, and the steps to run the demonstration software on the Nucleo-L476RG are given. This is the development board used in all the projects described in this book.

In the next Chapter we shall be looking at the STM32 processor architecture.

CHAPTER 2 • STM32 Nucleo Processor Architecture

2.1 Overview

In this Chapter we shall be looking at the internal architecture of the STM32 Nucleo processors. This is important for developing efficient programs and projects. The popular STM-32L476RGT6 microcontroller used in all the projects in this book will be considered as an example Arm processor.

2.2 Arm processors

The choice of a microcontroller for a particular application depends on many factors such as the following:

- cost
- speed
- power consumption
- size
- number of digital and analogue input/output ports
- digital input-output port current capacity
- analogue port resolution and accuracy
- program and data memory sizes
- interrupt support
- timer support
- USART support
- special bus support (e.g. USB, CAN, SPI, I²C, and so on)
- ease of system development (e.g. programming)
- working voltage

For example, if you need to develop a battery powered device such as a mobile phone or a games device, then very high clock speed as well as long battery life are main requirements. But if you are developing a traffic lights controller then very high performance is not a requirement. In general, as the clock speed goes up so does the power consumption and as a result a trade-off should be made in choosing a microcontroller for a specific application.

Arm has been designing 32-bit processors for over 20 years and in the last few years they have also started to offer 64-bit designs. In actual fact, Arm is a company specialized in designing the processor architecture and they do not manufacture or sell processor chips. Arm makes money by licensing their designs to chip manufacturers. The manufacturers use the core Arm processors (e.g. the core CPU) and integrate with their own peripherals to end up in a complete microcontroller chip. Arm is then given royalty fees for each chip manufactured by the third-party companies. Companies that use Arm core processors include Apple, Atmel, Broadcom, Cypress Semiconductors, Freescale Semiconductors, Analog Devices, Nvidia, NXP, Samsung Electronics, Texas Instruments, Qualcomm, Renesas, and many others.

Arm was originally known as the Acorn Computers and they have developed the first Acorn RISC Machine (Arm) architecture in the 1980s to use in their personal computers. First Arm processors were co-processor modules used in the BBC Micro series. After failing to find suitable high-performance microprocessor chips in the market, Acorn decided to design their own processors. In 1990, the research section of Acorn formed the ARM Ltd.

Currently Arm is the most widely used processor in terms of quantity manufactured. Over 50 billion Arm processors have been produced as of 2014, where 10 billion were produced in 2013 only. The Arm 32-bit architecture is the most widely used in mobile devices where about 98% of all mobile phones sold in the year 2005 used at least one Arm processor. The Arm architecture is known to offer the best MIPS-to-watts ratio as well as MIPS-to-\$ ratio in the industry, and the smallest CPU size.

The small size, low power consumption, and low cost make Arm an ideal processor in embedded applications. Arm processors are based on an instruction set called Thumb. With clever design this instruction set takes 32-bit instructions and compresses them down to 16-bits, thus reduces the hardware size, which also reduces the overall cost. The processor makes use of multistage pipelined architecture that is easier to learn, build, and program. Arm processors are based on the RISC (Reduced instruction Set Computer) architecture and are available as 32-bit or 64-bit multi-core structures. RISC processors, as opposed to CISC (Complex Instruction Set Computer) processors have smaller number of instructions and fewer transistors (hence smaller die size), as a result they can operate at higher speeds. Unimportant or not frequently used instructions are removed, and the pathways are optimized to result in superior performance.

It is important to realize that Arm's core architecture is only a processor and it does not include graphics, input-output ports, USB, serial communication, wireless connectivity, or any other form of peripheral modules. Chip manufacturers build their systems around the Arm core design and therefore different manufacturers offer different types of Arm based microcontrollers.

Over the last 20 years or so Arm (ARM) had developed many 32-bit processors. Figure 2.1 shows some of the popular members of the Arm processor family. The first successful member was the ARM7TDMI which had high code density and low power consumption. This processor, based on the Von Neumann architecture, was operating at 80 MHz and was used in early mobile phones. ARM9 was developed in 1997 with Harvard architecture and operated with 150 MHz, thus offered higher performance. ARM10 and ARM11 were developed in the years 1999 and 2003, respectively. Both processors were based on the Harvard architecture. ARM10 operated at 260 MHz and ARM11 operated at 335 MHz. Around the year 2003 Arm decided to improve their market share by developing new series of high-performance processors. As a result, the Cortex family of processors were created. The Cortex family consists of three processor families: Cortex-M, Cortex-R, and Cortex-A. We shall now briefly look at these families.

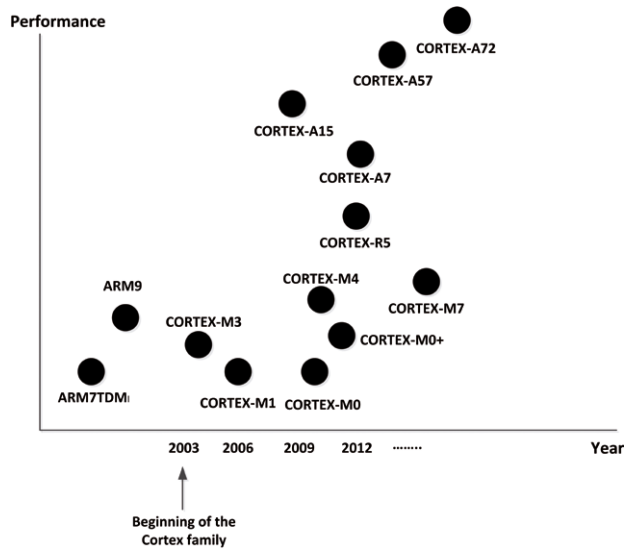


Figure 2.1: Overview of the Arm processor family.

2.2.1 Cortex-M

Cortex-M series are built around the ARMv6-M architecture (Cortex-M0 and Cortex-M0+) and the ARMv7-M architecture (Cortex-M3 and Cortex-M4). These processors are specifically designed for the microcontroller market, offering quick and deterministic interrupt responses, low power consumption, low cost, fairly high performance, and ease of use. The Cortex-M3 and Cortex-M4 are very similar in architecture and have the same instruction set (Thumb 2) with the difference that the Cortex-M4 offers digital signal processing (DSP) capability and has optional floating point unit (FPU). Cortex-M4 with its DSP and floating-point capability is an ideal processor for the IoT and wearable applications. For cost sensitive and lower performance applications the Cortex-M0 or the Cortex-M0+ can be used. The Cortex-M0 processor has small gate count (12K gates) and consumes only 12.5 $\mu\text{W}/\text{MHz}$. The Cortex-M0+ consumes only 9.85 $\mu\text{W}/\text{MHz}$ and is based on a subset of the Thumb 2 instruction set and its performance is slightly above that of Cortex-M0 and below that of the Cortex-M3 and Cortex-M4. Cortex-M7 is a high performance processor that can handle fast DSP and single or double precision floating point operations and is mainly used in applications where higher performance than the Cortex-M4 is required.

2.2.2 Cortex-R

Cortex-R series are real-time higher performance processors than the Cortex-M and some members are designed to operate at high clock rates in excess of 1 GHz. These processors are commonly used in hard-disk controllers, network devices, automotive applications, and in specialized high-speed microcontroller applications. Cortex-R4 and Cortex-R5 are the early members and can be used at clock speeds of up to 600 MHz. Cortex-R7 is a newer member that incorporates 11-stage pipeline for high performance, and it can operate in excess of 1 GHz. Although the Cortex-R processors are high performance their architecture is complex and these processors consume high power, making them unsuitable for use in mobile battery powered devices.

2.2.3 Cortex-A

Cortex-A are the highest performance Arm processors designed for use with real-time operating systems in mobile applications such as in mobile phones, tablets, GPS devices and so on. These processors support advanced features for operating systems such as Android, iOS, Linux, Windows etc. In addition, advanced memory management is supported with virtual memory. Early members of the family included processors such as Cortex-A5 to Cortex-A17, based on the ARMv7-A architecture. Latest members of the family are the Cortex-A50 and Cortex-A72 series designed for low power and very high-performance mobile applications. These processors are built using the ARMv8-A architecture which offers 64-bit energy-efficient operation with the capability of more than 4 GB of physical memory.

2.2.4 Cortex-M processor comparison

A comparison of the various Cortex-M series processors is given in Table 2.1. As can be seen from this table, Cortex-M0 and Cortex-M0+ are used at low speed and low power consumption applications. Cortex-M1 is optimized for use in programmable gate array applications. Cortex-M3 and Cortex-M4 are medium power processors used in microcontroller applications with the Cortex-M4 supporting DSP and floating-point arithmetic operations. Cortex-M7 is a high-performance member of the family which is used in applications requiring higher performance than the Cortex-M4.

Processor	Description
Cortex-M7	High performance processor, used in applications where Cortex-M4 is not fast enough, supports DSP and single and double precision arithmetic
Cortex-M4	Similar architecture as the Cortex-M3 but includes DSP and floating point arithmetic, used in high-end microcontroller type applications
Cortex-M3	Very popular, low power consumption, medium performance, debug features, used in microcontroller type applications
Cortex-M1	Designed mainly for programmable gate array applications
Cortex-M0+	Lower power consumption and higher performance than the Cortex-M0
Cortex-M0	Low power consumption, low to medium performance, smallest Arm processor

Table 2.1: Cortex-M processor comparison.

2.2.5 Processor performance measurement

Processor performance is usually measured using benchmark programs. There are many benchmark programs available and one should exercise care when comparing the performance of various processors as the performance depends upon many external factors such as the efficiency of the compiler used and the type of operation performed for the measurement.

Many attempts were made in the past to measure the performance of a processor and quote it as a single number. For example, MOPS, MFLOPS, Dhrystone, DMIPS, BogoMIPS, and so on. Nowadays, CoreMark is one of the most commonly used benchmark programs used to indicate the processor performance. CoreMark is developed by Embedded Microprocessor Benchmark Consortium (EEMBC, www.eembc.org/coremark) and is one of the most reliable performance measurement tools available.

Table 2.2 shows the CoreMark results for some of the commonly used microcontrollers. As can be seen from this table, Cortex-M7 achieves 5.01 CoreMark/MHz, while the PIC18 microcontroller achieves only 0.04 CoreMark/MHz.

Processor	CoreMark/MHz
Intel Zeus	8.01
Cortex-M7	5.01
Cortex-A9	4.15
Espressif ESP32	4.13
Cortex-M4	3.40
Cortex-M3	3.32
Cortex-M0+	2.49
Espressif ESP8266	2.38
Cortex-M0	2.33
dsPIC33	1.89
PIC24	1.88
MSP430	1.11
PIC18	0.04

Table 2.2: CoreMark/MHz for some commonly used microcontrollers.

2.2.6 Cortex-M compatibility

Processors in the Cortex family are upward compatible with each other. Cortex-M0 and Cortex-M0+ processors are based on the ARMv6-M architecture, using the Thumb instruction set. On the other hand, Cortex-M3, Cortex-M4 and Cortex-M7 are based on the ARMv7-M architecture, using the Thumb 2 instruction set which is a superset of the Thumb instruction set. Although the architectures are different, software developed on the Cortex-M0 and Cortex-M0+ processors can run on the Cortex-M3, Cortex-M4, and Cortex-M7 processors without any modifications provided the required memory and input-output ports are available.

2.2.7 Choice of an STM32 processor

Developers may find it difficult to choose the correct STM32 processor for the correct project. STMicroelectronics offers a MCU selection tool which makes it easy to choose the processor with the features you are interested in. This tool is available at the following website: <http://www.st.com/web/en/catalog/mmc/FM141/SC1169>

2.3 The STM32L476RGT6 microcontroller

In this book we shall be using the highly popular Arm microcontroller STM32L476RGT6 in the STM Nucleo-L476RG development board. In this Chapter we shall be looking at the features of the STM32L476RGT6 microcontroller. This microcontroller type STM32L476 belongs to the family 'STM32L476xxxx' where R is for 64 pins, G is for 1 MB flash memory, T is for LQPF package, and 6 is for industrial temperature range -40° to $+85^{\circ}\text{C}$. The internal architecture of this microcontroller is very complex and we shall only look at the important

modules used in most projects, such as I/O, timers, ADC converter and DAC converter, interrupts, I²C, USART, and so on. Interested readers can get detailed information from the manufacturer's datasheets available for download on the Internet. Readers should note that the structure of the STM32 family of processors is similar and after becoming familiar with the STM32L476RGT6, it should not be too difficult to learn the structure of the others in the family.

2.3.1 Basic features of the STM32L476RGT6

The STM32L476RGT6 microcontroller is based on the Cortex-M4 architecture and has the following basic features (Further information can be obtained from the ST datasheet STM-32L476xx, DocID025976, 2017):

- Arm Cortex-M4 32-bit CPU with FPU
- 80 MHz max CPU frequency
- Operating voltage 1.71 to 3.6 V
- 1.25 DMIPS/MHZ performance (Drystone 2.1)
- 4 to 48 MHz crystal oscillator
- 32 MHz crystal oscillator for RTC
- Internal 16 MHz RC oscillator
- Internal 32 kHz RC oscillator
- Internal 100 kHz to 48 MHz oscillator
- 3 PLLs for system clock
- 1 MB flash memory
- 128 KB SRAM
- 16× timers
- 3× SPI and 3× I²C interfaces
- 3× USART, 2× UART, 1× LPUART interfaces
- 2× SAI and 1× CAN interfaces
- 4× digital filters for sigma-delta modulators
- 1× Real Time Clock (RTC)
- LCD interface (8 × 28 or 4 × 32)
- 51× General Purpose I/O (GPIOs)
- 12× capacitive sensing channels
- 3× 16 channel 12-bit ADC
- 2× 12-bit DAC
- 14 channel DMA
- Temperature sensor
- Random number generator
- Cyclic Redundancy Check (CRC)
- USB, SDMMC and SWPMI communication interface
- 2× analogue comparator and 2× operational amplifiers
- 64-pin package

2.3.2 Internal block diagram

Figure 2.2 shows the functional block diagram of the STM32L476xx processor. The 80 MHz Arm Cortex-M4 CPU is shown at the middle top of the figure. The digital connectivity modules are at the top left, the GPIOs are at the bottom left, the LCD interface is at the top

right, timers are shown at the middle right, analogue modules and parallel interface are at the bottom right, and the DMA channels and the memory are shown at the middle part of the figure. Notice that this figure is not specific to the STM32L476RGT6 microcontroller, but it refers to the STM32L476xx family.

Figure 2.3 shows the pin configuration of the STM32L476Rx. The pin definitions are as follows (some of the pins are shared with other functions):

PA0 – PA15	-	GPIO Port A pins
PB0 – PB15	-	GPIO Port B pins
PC0 – PC15	-	GPIO Port C pins
PD2	-	GPIO Port D pin
PH0 – PH1	-	GPIO Port H pins
VDD, VSS	-	Power and ground pins
VSSA, VDDA	-	Reference voltage pins
NRST	-	Reset pin
VBAT	-	External battery pin
BOOT0	-	Boot0 pin
VDD12	-	External SMPS power pin

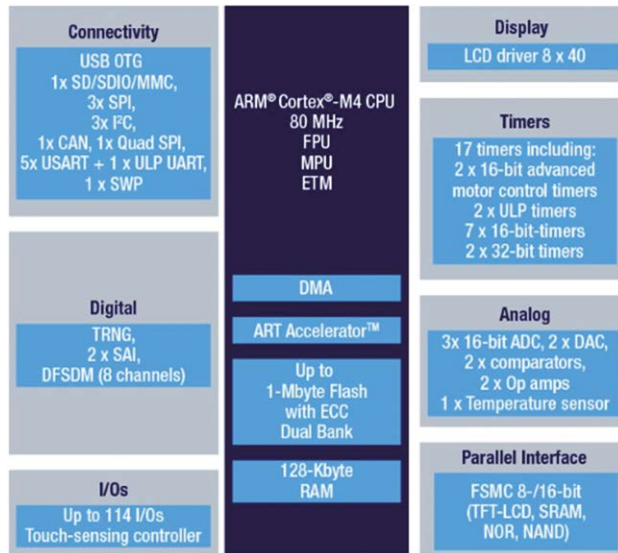


Figure 2.2: Functional block diagram of the STM32L476xx processor.

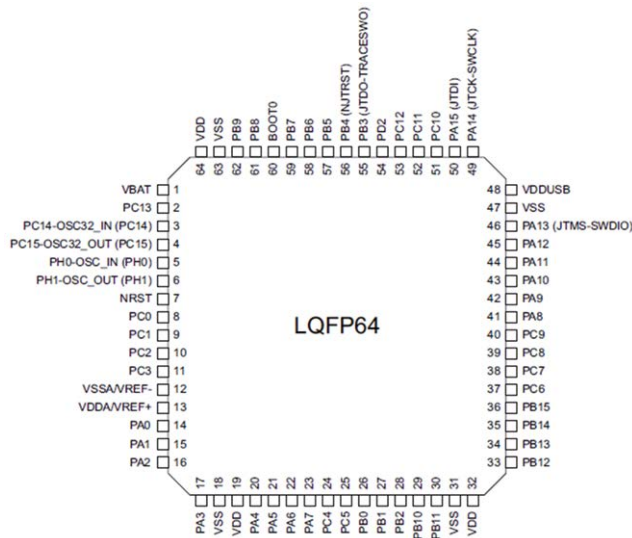


Figure 2.3: Pin configuration of the STM32L476Rx.

A simplified internal block diagram of the STM32L476Rx processor is shown in Figure 2.4. At the top left corner is the 80 MHz Cortex-M4 processor with the flash memory and SRAM on the right-hand side of the processor. The DMA channels are below the processor. There are four major busses running inside the chip, called the AHB1, AHB2, APB1 and APB2. The GPIO ports and the ADC are connected to the 80 MHz AHB2 bus. The SDIO/MMC, Timers 1,8,15,16, 17, SPI1, SAI1 and SAI2, and the comparators are connected to the APB2 bus. Timers 2,3,4,5,6,7, USART 2,3, UART 4,5, SP2 and SP3, and I2C1,2,3, CAN, LCD driver, LPUART1, LPTIM1,2 DAC 1,2 and the watchdog timer are connected to the APB1 bus.

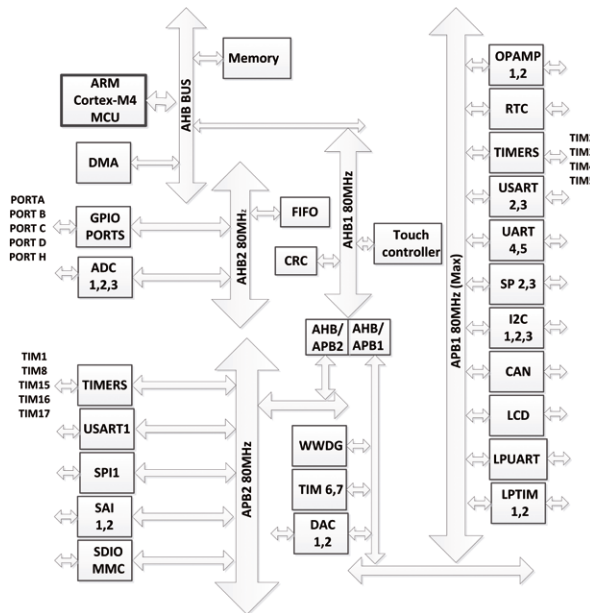


Figure 2.4: Simplified internal structure of the STM32L476Rx microcontroller.

2.3.3 General purpose inputs and outputs (GPIOs)

The STM32L476Rx microcontroller has 64 pins and 51 of them can be used as general-purpose inputs or outputs (GPIOs). The GPIO is arranged into 5 ports, where ports A, B, and C are 16-bits wide, port H is 2-bits wide, and port D is only 1-bit. During and after reset most ports are configured in analogue input mode. Each port has the following basic features:

- all GPIO pins have weak internal pull-up and pull-down resistors, which can be activated under software control;
- port input states can be floating, pull-up, pull-down, or analogue;
- port outputs can be configured as push-pull, open-drain or pull-up, pull-down;
- the speed of each port can be set;
- I/O port configurations can be locked under software control;
- port pins can be digital I/O, analogue input, or they can have alternative functions, such as DAC, SPI, USB, PWM, etc.);
- each port pin can be used with one of 15 alternate functions;
- bit manipulations can be performed on each port pin.

Each GPIO port, subject to its hardware characteristics, can be configured in software in several modes:

- input pull-up;
- input-pull-down;
- analogue;
- output open-drain with pull-up or pull-down capability;
- output push-pull with pull-up or pull-down capability;
- alternate function push-pull with pull-up or pull-down capability;
- alternate function open-drain with pull-up or pull-down capability.

The I/O port registers must be accessed as 32-bit words, half-words or bytes. Each I/O port has 4× 32 bit configuration registers:

GPIOx_MODER, GPIOx_OTYPER, GPIOx_OSPEEDR, and GPIOx_PUPDR

2× 32 bit data registers:

GPIOx_IDR and GPIOx_ODR

1× 32 bit set/reset register:

GPIOx_BSRR

1× 32 bit locking register:

GPIOx_LCKR

and 2× 32 bit alternate function selection registers:

GPIOx_AFRH and GPIOx_AFRL.

Set/reset registers are used for read/modify accesses to any port pin without being interrupted by the system interrupt controller module.

Figure 2.5 shows the structure of a push-pull output port pin. Similarly, an open-drain output port pin is shown in Figure 2.6. Input pull-up and pull-down circuits are shown in Figure 2.7 and Figure 2.8, respectively.

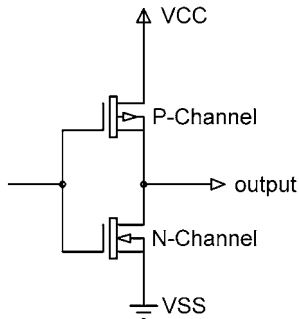


Figure 2.5: Push-pull output pin.

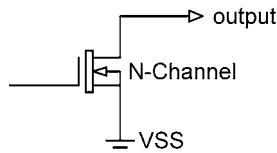


Figure 2.6: Open-drain output pin.

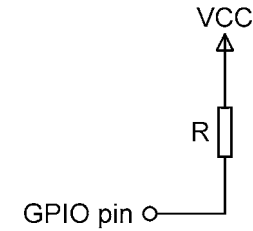


Figure 2.7: Pull-up pin.

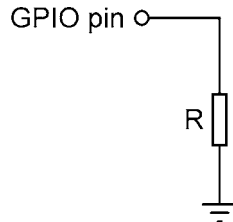


Figure 2.8: Pull-down pin.

The basic structure of an I/O port pin is shown in Figure 2.9. The push-pull transistors and pull-up and pull-down resistors can be seen in the figure. Notice that protection diodes are used at the port inputs to protect the input circuitry from high voltages.

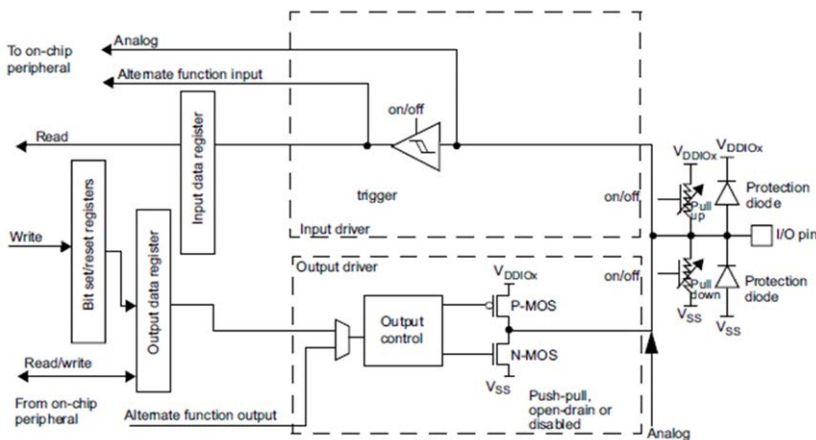


Figure 2.9: Structure of an I/O pin.

Port data registers GPIOx_IDR (Figure 2.10) and GPIOx_ODR (Figure 2.11) are used to read and write data to a port, respectively. When the pin is configured as output, the value written to the output data register GPIOx_ODR is output on the I/O pin. It is possible to use the output driver in push-pull mode or open-drain mode (only the low level is driven, high level is HI-Z). The input data register GPIOx_IDR captures the data present on the I/O pin at every AHB clock cycle.

Address offset: 0x10

Reset value: 0x0000 XXXX (where X means undefined)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ID15	ID14	ID13	ID12	ID11	ID10	ID9	ID8	ID7	ID6	ID5	ID4	ID3	ID2	ID1	ID0
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

Bits 31:16: Reserved, must be kept at reset value.

Bits 15:0 **IDy**: Port input data bit (y = 0..15)

These bits are read-only. They contain the input value of the corresponding I/O port.

Figure 2.10: GPIOx_IDR port input register.

Address offset: 0x14

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OD15	OD14	OD13	OD12	OD11	OD10	OD9	OD8	OD7	OD6	OD5	OD4	OD3	OD2	OD1	OD0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:16: Reserved, must be kept at reset value.

Bits 15:0 **ODy**: Port output data bit (y = 0..15)

These bits can be read and written by software.

Note: For atomic bit set/reset, the OD bits can be individually set and/or reset by writing to the GPIOx_BSRR or GPIOx_BRR registers (x = A..F).

Figure 2.11: GPIOx_ODR port output register.

All GPIO pins have weak internal pull-up and pull-down resistors, which can be activated or not depending on the value in the GPIOx_PUPDR register (Figure 2.12).

Address offset: 0x0C

Reset values:

- 0x6400 0000 for port A
- 0x0000 0100 for port B
- 0x0000 0000 for other ports

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
PUPD15[1:0]	PUPD14[1:0]	PUPD13[1:0]	PUPD12[1:0]	PUPD11[1:0]	PUPD10[1:0]	PUPD9[1:0]	PUPD8[1:0]								
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PUPD7[1:0]	PUPD6[1:0]	PUPD5[1:0]	PUPD4[1:0]	PUPD3[1:0]	PUPD2[1:0]	PUPD1[1:0]	PUPD0[1:0]								
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 2y+1:2y **PUPDy[1:0]**: Port x configuration bits (y = 0..15)

These bits are written by software to configure the I/O pull-up or pull-down

- 00: No pull-up, pull-down
- 01: Pull-up
- 10: Pull-down
- 11: Reserved

Figure 2.12: GPIOx_PUPDR pull-up/pull-down register.

The Port Bit Set/Reset Register `GPIOx_BSRR` (Figure 2.13) is used in read/modify operations. Writing any bit to 0 in `GPIOx_BSRR` does not have any effect on the corresponding bit in `GPIOx_ODR`. If there is an attempt to both set and reset a bit in `GPIOx_BSRR`, the set action takes priority. Using the `GPIOx_BSRR` register to change the values of individual bits in `GPIOx_ODR` is a ‘one-shot’ effect that does not lock the `GPIOx_ODR` bits. The `GPIOx_ODR` bits can always be accessed directly. The `GPIOx_BSRR` register provides a way of performing atomic bitwise handling. Using these registers to modify a port pin protects the operation from interrupts. Thus, there is no need for the software to disable interrupts during port read/write operations.

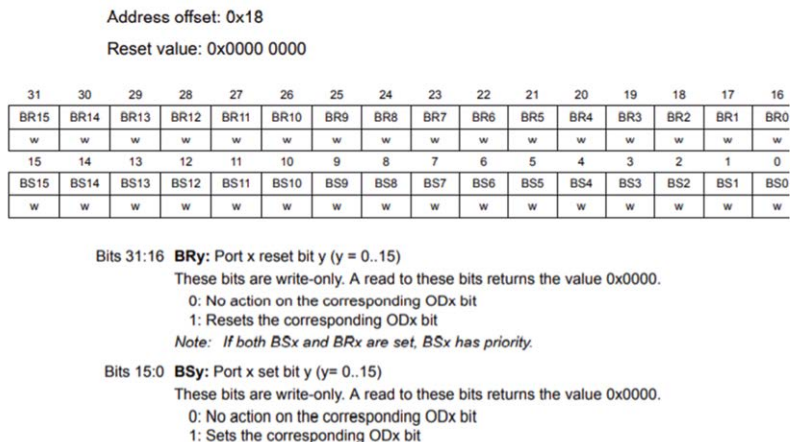


Figure 2.13: `GPIOx_BSRR` bit set/reset register.

Port locking register `GPIOx_LCKR` (Figure 2.14) allows the input-output configuration of a port to be frozen until the microcontroller is reset. The locking operation is useful when the ports are configured, and it is required to protect this configuration from accidental changes.

Address offset: 0x1C

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	LCKK
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LCK15	LCK14	LCK13	LCK12	LCK11	LCK10	LCK9	LCK8	LCK7	LCK6	LCK5	LCK4	LCK3	LCK2	LCK1	LCK0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:17 Reserved, must be kept at reset value.

Bit 16 LCKK: Lock key

This bit can be read any time. It can only be modified using the lock key write sequence.

0: Port configuration lock key not active

1: Port configuration lock key active. The GPIOx_LCKR register is locked until the next MCU reset or peripheral reset.

LOCK key write sequence:

WR LCKR[16] = '1' + LCKR[15:0]

WR LCKR[16] = '0' + LCKR[15:0]

WR LCKR[16] = '1' + LCKR[15:0]

RD LCKR

RD LCKR[16] = '1' (this read operation is optional but it confirms that the lock is active)

Note: During the LOCK key write sequence, the value of LCK[15:0] must not change.

Any error in the lock sequence aborts the lock.

After the first lock sequence on any bit of the port, any read access on the LCKK bit will return '1' until the next MCU reset or peripheral reset.

Bits 15:0 LCKy: Port x lock bit y (y= 0..15)

These bits are read/write but can only be written when the LCKK bit is '0'.

0: Port configuration not locked

1: Port configuration locked

Figure 2.14: GPIOx_LCKR port locking register.

The frozen registers are: GPIOx_MODER, GPIOx_OTYPER, GPIOx_OSPEEDR, GPIOx_PUPDR, GPIOx_AFR1, and GPIOx_AFRH. When the LOCK sequence has been applied to a port bit, the value of the port bit can no longer be modified until the next MCU reset or peripheral reset. Each GPIOx_LCKR bit freezes the corresponding bit in the control registers GPIOx_MODER, GPIOx_OTYPER, GPIOx_OSPEEDR, GPIOx_PUPDR, GPIOx_AFR1, and GPIOx_AFRH.

Port configuration register GPIOx_MODER (Figure 2.15) configures a port in input mode, output mode, alternate function mode, or in analogue mode.

Address offset: 0x00

Reset values:

- 0xABFF FFFF for port A
- 0xFFFF FEBF for port B
- 0xFFFF FFFF for ports C..G, I
- 0x0000 000F for port H

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MODE15[1:0]		MODE14[1:0]		MODE13[1:0]		MODE12[1:0]		MODE11[1:0]		MODE10[1:0]		MODE9[1:0]		MODE8[1:0]	
r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MODE7[1:0]		MODE6[1:0]		MODE5[1:0]		MODE4[1:0]		MODE3[1:0]		MODE2[1:0]		MODE1[1:0]		MODE0[1:0]	
r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w

Bits 2y+1:2y **MODEy[1:0]**: Port x configuration bits (y = 0..15)

These bits are written by software to configure the I/O mode.

00: Input mode

01: General purpose output mode

10: Alternate function mode

11: Analog mode (reset state)

Figure 2.15: GPIOx_MODER port configuration register.

Register GPIOx_OTYPER (Figure 2.16) configures a port in output push-pull (reset state) or in open-drain.

Address offset: 0x04

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OT15	OT14	OT13	OT12	OT11	OT10	OT9	OT8	OT7	OT6	OT5	OT4	OT3	OT2	OT1	OT0
r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w

Bits 31:16 Reserved, must be kept at reset value.

Bits 15:0 **OTy**: Port x configuration bits (y = 0..15)

These bits are written by software to configure the I/O output type.

0: Output push-pull (reset state)

1: Output open-drain

Figure 2.16: GPIOx_OTYPER port configuration register.

Register GPIOx_OSPEEDR (Figure 2.17) configures the port speed as low, medium, high, or very high. It is important to understand that the port speed does not refer to the performance (i.e. the switching frequency) of a port pin, instead, it defines the slew rate of a GPIO pin. i.e. how fast the port pin goes from logic 0 to logic 1, and *vice versa*.

Address offset: 0x08

Reset value:

- 0x0C00 0000 for port A
- 0x0000 0000 for the other ports

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
OSPEED15 [1:0]		OSPEED14 [1:0]		OSPEED13 [1:0]		OSPEED12 [1:0]		OSPEED11 [1:0]		OSPEED10 [1:0]		OSPEED9 [1:0]		OSPEED8 [1:0]	
r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OSPEED7 [1:0]		OSPEED6 [1:0]		OSPEED5 [1:0]		OSPEED4 [1:0]		OSPEED3 [1:0]		OSPEED2 [1:0]		OSPEED1 [1:0]		OSPEED0 [1:0]	
r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w

Bits 2y+1:2y OSPEEDy[1:0]: Port x configuration bits (y = 0..15)

These bits are written by software to configure the I/O output speed.

00: Low speed

01: Medium speed

10: High speed

11: Very high speed

Figure 2.17: GPIOx_OSPEEDR port speed register.

When the I/O port is programmed as analogue configuration:

- the output buffer is disabled;
- the Schmitt trigger input is deactivated;
- the weak pull-up and pull-down resistors are disabled by hardware;
- read access to the input data register gets the value '0'.

Port pins can be programmed for alternate functions (AF). For alternate function inputs the port must be configured in the required input mode. Similarly, for alternate function outputs, the port must be configured in alternate function output mode. When a port is configured as alternate function, the pull-up and pull-down resistors are disabled, the output is set to operate in push-pull or in open drain mode. Registers GPIOx_AFRL (Figure 2.18) and GPIOx_AFRH (Figure 2.19) configure the GPIO alternate function low and high registers. When an I/O port is configured as alternate function:

- the output buffer can be configured in open-drain or push-pull mode;
- the output buffer is driven by the signals coming from the peripheral;
- the Schmitt trigger input is activated;
- the state of the pull-up and pull-down resistors depend on the value in register GPIOx_PUPDR;
- the data present on the I/O pin are readapt every AHB clock cycle.

Address offset: 0x20

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
AFSEL7[3:0]				AFSEL6[3:0]				AFSEL5[3:0]				AFSEL4[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
AFSEL3[3:0]				AFSEL2[3:0]				AFSEL1[3:0]				AFSEL0[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:0 **AFSELY[3:0]**: Alternate function selection for port x pin y (y = 0..7)

These bits are written by software to configure alternate function I/Os

AFSELY selection:

0000: AF0	1000: AF8
0001: AF1	1001: AF9
0010: AF2	1010: AF10
0011: AF3	1011: AF11
0100: AF4	1100: AF12
0101: AF5	1101: AF13
0110: AF6	1110: AF14
0111: AF7	1111: AF15

Figure 2.18: GPIOx_AFRL register.

Address offset: 0x24

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
AFSEL15[3:0]				AFSEL14[3:0]				AFSEL13[3:0]				AFSEL12[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
AFSEL11[3:0]				AFSEL10[3:0]				AFSEL9[3:0]				AFSEL8[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:0 **AFSELY[3:0]**: Alternate function selection for port x pin y (y = 8..15)

These bits are written by software to configure alternate function I/Os

AFSELY selection:

0000: AF0	1000: AF8
0001: AF1	1001: AF9
0010: AF2	1010: AF10
0011: AF3	1011: AF11
0100: AF4	1100: AF12
0101: AF5	1101: AF13
0110: AF6	1110: AF14
0111: AF7	1111: AF15

Figure 2.19: GPIOx_AFRH register.

2.3.4 Electrical characteristics

It is important to know the absolute maximum and typical ratings of a microcontroller before it is used in a project. Stresses above the absolute maximum ratings for extended periods may affect device reliability and may even damage the device. The absolute voltage and current characteristics of the of the STM32L476xx microcontroller are shown in Table 2.3 and Table 2.4, respectively. Notice from this table that the output current sunk and sourced by any I/O pin is specified as 20 mA maximum and thus, direct LED-drive is possible. To drive larger loads, it will be necessary to use transistor switching circuits or relays. The total current sourced from all I/O pins plus the run consumption of the CPU cannot exceed 100 mA. Care should be taken when driving CMOS circuits that operate at +5 V since the output voltage of an I/O pin will not be high enough to drive a CMOS input, even with a pull-up resistor. In such circumstances 3 V to 5 V converter circuits (e.g. transistor switches or voltage converter integrated circuits) should be used.

Symbol	Ratings	Min	Max	Unit
$V_{DDX} - V_{SS}$	External main supply voltage (including V_{DD} , V_{DDA} , V_{DDIO2} , V_{DDUSB} , V_{LCD} , V_{BAT})	-0.3	4.0	V
$V_{DD12} - V_{SS}$	External SMPS supply voltage	Range 1	1.32	V
		Range 2		
$V_{IN}^{(2)}$	Input voltage on FT_XXX pins	$V_{SS}-0.3$	$\min(V_{DD}, V_{DDA}, V_{DDIO2}, V_{DDUSB}, V_{LCD}) + 4.0^{(3)(4)}$	V
	Input voltage on TT_XX pins	$V_{SS}-0.3$	4.0	
	Input voltage on BOOT0 pin	V_{SS}	9.0	
	Input voltage on any other pins	$V_{SS}-0.3$	4.0	

Table 2.3: Absolute voltage ratings © STMicroelectronics. Used with permission.

Symbol	Ratings	Max	Unit
ΣI_{VDD}	Total current into sum of all V_{DD} power lines (source) ⁽¹⁾⁽²⁾	150	mA
ΣI_{VSS}	Total current out of sum of all V_{SS} ground lines (sink) ⁽¹⁾	150	
$I_{VDD(PIN)}$	Maximum current into each V_{DD} power pin (source) ⁽¹⁾⁽²⁾	100	
$I_{VSS(PIN)}$	Maximum current out of each V_{SS} ground pin (sink) ⁽¹⁾	100	
$I_{IO(PIN)}$	Output current sunk by any I/O and control pin except FT_f	20	
	Output current sunk by any FT_f pin	20	
	Output current sourced by any I/O and control pin	20	
$\Sigma I_{IO(PIN)}$	Total output current sunk by sum of all I/Os and control pins ⁽³⁾	100	
	Total output current sourced by sum of all I/Os and control pins ⁽³⁾	100	
$I_{INJ(PIN)}^{(4)}$	Injected current on FT_XXX, TT_XX, RST and B pins, except PA4, PA5	-5/+0 ⁽⁵⁾	
	Injected current on PA4, PA5	-5/0	
$\Sigma I_{INJ(PIN)} $	Total injected current (sum of all I/Os and control pins) ⁽⁶⁾	25	

Table 2.4: Absolute current ratings © STMicroelectronics. Used with permission.

2.3.5 The power supply

The microcontroller is powered from its VDD pins with a voltage in the range of 1.71 V to 3.6 V. VDDA is the external analogue power supply for the ADC, DAC, and the analogue comparators and some other parts of the chip. A minimum of 1.62 V must be applied to VDDA when the ADC converter and the analogue comparators are used. The minimum voltage when the DAC and operational amplifiers are used is 1.8 V. Maximum VDDA voltage must not exceed 3.6 V. VDDA and VSSA can be connected to VDD and VSS respectively. VDDUSB is the external power supply for USB transceivers. This voltage must be in the range 3.0 V to 3.6 V. VBAT is the external battery voltage for the RTC, 32 kHz oscillator and backup registers when the VDD is not present. This voltage must be in the range 1.55 V to 3.6 V. VDD12 is the external power supply bypassing internal regulator when connected to an external SMPS and it must be between 1.05 V and 1.32 V. Figure 2.20 shows the power supply overview.

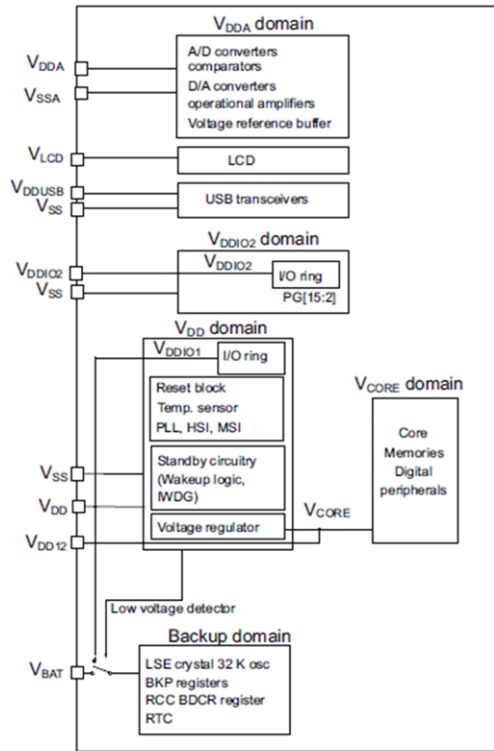


Figure 2.20: Power supply overview.

2.3.6 Low power modes

By default, the microcontroller is in Run mode after a Reset. The microcontroller can be configured to operate in one of three modes in low-power operations:

Sleep mode: In this mode the CPU is stopped, but all peripherals continue to operate. The CPU wakes up when an interrupt occurs.

Low-power run mode: In this mode the code can be executed from SRAM or from Flash, and the CPU frequency is limited to 2 MHz. The peripherals with independent clock can be clocked by HSI16.

Low-power sleep mode: This mode is entered from the low-power run mode. Only the CPU clock is stopped. When wakeup is triggered by an event or an interrupt, the system reverts to the low-power run mode.

Stop mode: In this mode all clocks are stopped, and this mode provides the lowest power consumption while retaining the contents of the CPU registers and SRAM. Three Stop modes are available: Stop 0, Stop 1 and Stop 2. In Stop 0 mode, the main regulator remains ON, allowing a very fast wakeup time but with much higher consumption. Stop 1 offers the largest number of active peripherals and wakeup sources, a smaller wakeup time

but a higher consumption. In Stop 2 mode, most of the VCORE domain is put in a lower leakage mode. The system clock when exiting from Stop 0, Stop 1 or Stop 2 modes can be either MSI up to 48 MHz or HSI16, depending on software configuration.

Standby mode: In this mode the oscillators and the voltage regulator are switched off and thus it provides the lowest power consumption. All register contents and SRAM contents are lost (except for registers in the Backup circuitry). The device exits Standby mode when an external reset, a watchdog reset, wake-up pin event, or an RTC event occurs, or a failure is detected on LSE. The system clock after wakeup is MSI up to 8 MHz.

Shutdown mode: The Shutdown mode achieves the lowest power consumption. The internal regulator is switched off so that the VCORE domain is powered off. The PLL, the HSI16, the MSI, the LSI and the HSE oscillators are also switched off. The device exits Standby mode when an external reset, a watchdog reset, wake-up pin event, or an RTC event occurs. The system clock after wakeup is MSI up to 4 MHz.

2.3.7 The clock circuit

The clock circuit of the STM32L476xx microcontroller is very powerful and at the same time very complex. In this section we shall be looking at the various clock options and how to configure the clock.

Basically, there are two types of system clock (SYSCLK) sources (see Figure 2.21): *External* clock sources and *Internal* clock sources. Figure 2.22 and Figure 2.23 show the block diagram of the clock circuit.

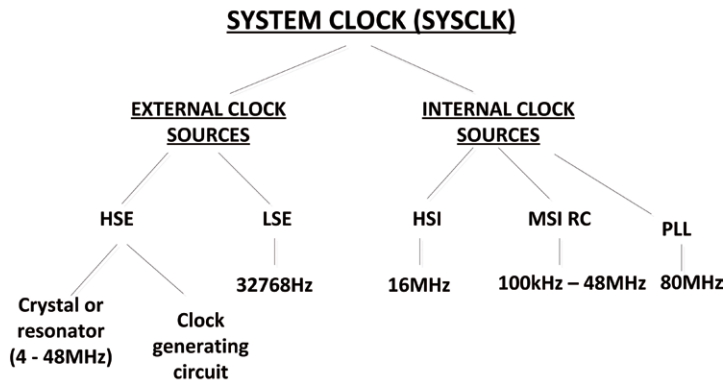


Figure 2.21: STM32L476xx microcontroller system clock sources.

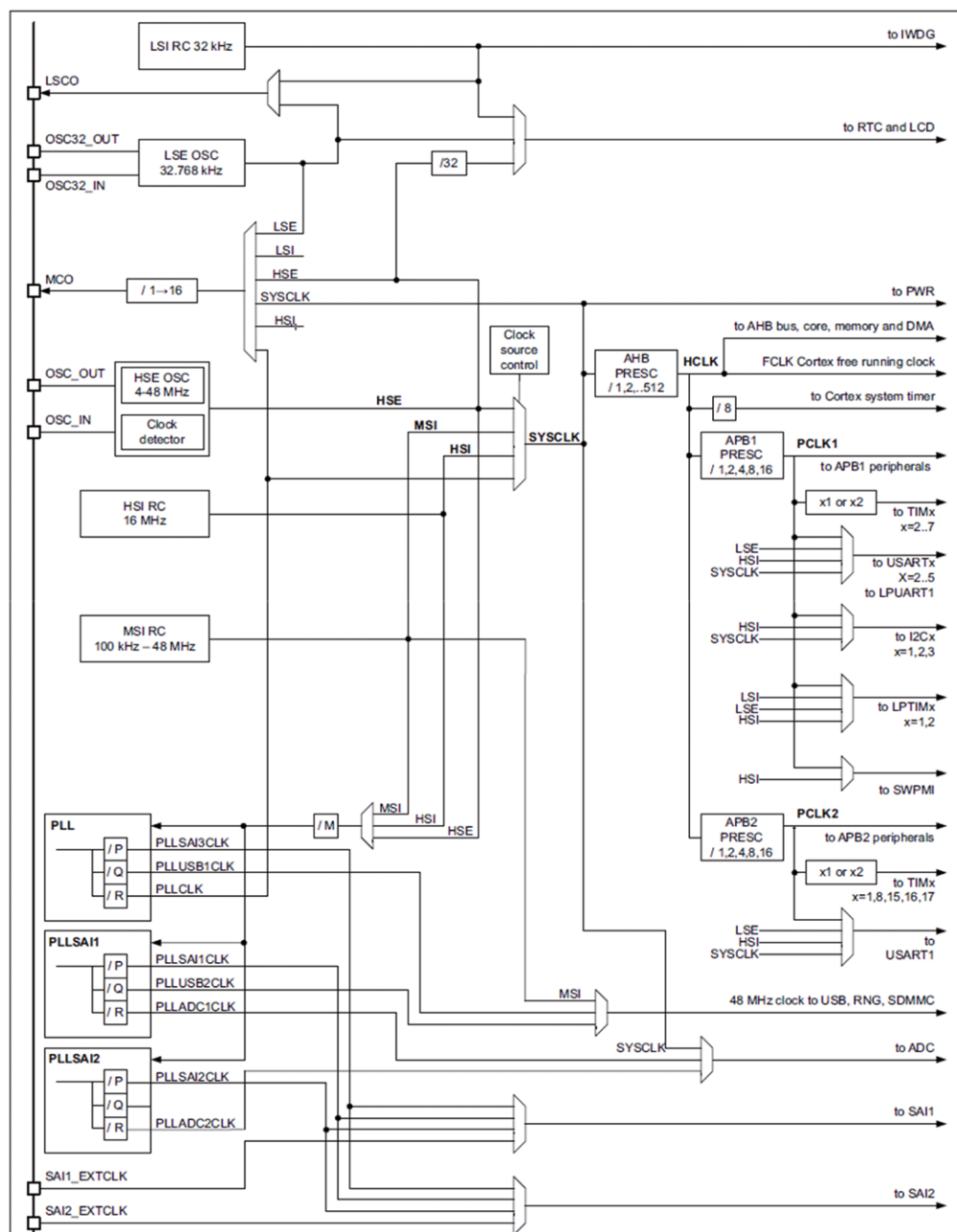


Figure 2.22 / 2.23: STM476xx clock circuit.

External clock sources

High Speed External (HSE): This can be an external crystal or resonator device, or an external clock signal. The frequency range of the crystal or resonator should be 4–48 MHz.

Figure 2.24 shows a typical crystal connection. It is recommended to use two capacitors in the range of 4–25pF with the quartz crystal circuit.

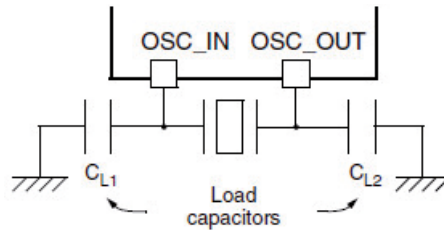


Figure 2.24: Crystal oscillator connection.

When using a clock generator circuit, the waveform can be square, sine, or triangular, and the waveform must be symmetrical, i.e. 50% ON and 50% OFF times. The clock signal must be fed to the OSC_IN pin of the microcontroller (Figure 2.25).

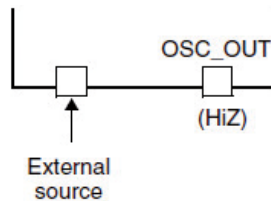


Figure 2.25: Using clock generator circuit.

If external clock circuitry is used, then the HSE oscillator should be bypassed to avoid any conflict.

Low Speed External (LSE): This is a 32,768 Hz clock driven from an external crystal and feeding the internal Real Time Clock (RTC) module.

Internal Clock Sources

High Speed Internal (HSI): This is an accurate RC based 16 MHz internal clock with a factory calibrated tolerance of 1%.

Multispeed Internal (MSI RC): This is a multispeed RC based clock source providing clock in the range 100 kHz to 48 MHz. This clock can be trimmed by software and is able to generate 12 different clock frequencies.

PLL: The PLL is fed by HSE, HIS or MSI clocks and it can generate system clocks up to 80 MHz.

Microcontroller Clock Output (MCO)

A clock output is possible from a special pin called Microcontroller Clock Output (MCO). This clock output can be used as a general-purpose clock or as a clock for another microcontroller.

Low Speed Clock Output (LSCO)

A low-speed (32.768 kHz) clock (LSCO) is available as a general-purpose clock output. Several prescalers are available in the clock circuit to configure the AHB, APB1 and APB2 bus clock frequencies for the system and the peripheral devices. The maximum frequency of the AHB and APBx is 80 MHz.

Configuring the Clock

As shown in Figure 2.22 and Figure 2.23, the clock circuit consists of a number of multiplexers, prescalers, and a phase locked loop (PLL). The multiplexers are used to select the required clock source. The prescalers are used to divide the clock frequency by a constant. Similarly, the PLL is used to multiply the clock frequency with a constant to operate the chip at higher frequencies.

It is important to select the correct clock source for an application. Configuring the clock sources by programming the internal clock registers is a complex task and detailed knowledge of the clock circuitry is required. Referring to Figure 2.22 and Figure 2.23, we can identify the following prescalers:

AHB prescaler: this prescaler receives the system clock SYSCLK and provides clock to the AHB bus (HCLK). The prescaler can take the values from 1, 2, up to 512.

APB1 prescaler: this prescaler receives the AHB clock (HCLK) and provides clock to the APB1 bus peripherals (see Figure 2.4). The prescaler can take the values 1, 2, 4, 8, and 16.

APB2 prescaler: this prescaler receives the AHB clock (HCLK) and provides clock to the APB2 bus peripherals (see Figure 2.4). The prescaler can take the values 1, 2, 4, 8, and 16.

2.3.8 Analogue to digital converter (ADC)

The STM32L476xx microcontroller includes three successive-approximation ADC modules with up to 24 channels, with the following basic features:

- 12-bit resolution
- 5.33 megasamples (MS) per second conversion rate
- Internal and external reference voltage selection
- Single-ended and differential inputs
- Dual clock operation with the ADC speed independent of the CPU frequency
- Single-shot or continuous operation
- Results stored in three data registers (or in RAM)
- Programmable sampling time
- Generating interrupts and trigger for selected timers
- Data pre-processing (left/right alignment) and offset compensation

2.3.9 Digital to analogue converter (DAC)

Two DACs are provided with the following basic features:

- 8 or 12-bit output
- Data alignment (left or right) in 12-bit mode
- Noise-wave and triangular-wave generation
- DMA capability for each channel

- Dual independent DAC operation
- Triggering through the timer update outputs as well as based on external trigger
- Sample and hold (with internal or external capacitor)

2.3.10 Timers

The STM32L476xx includes two advanced control timers, up to nine general-purpose timers, two basic timers, two low-power timers, two watchdog timers and a SysTick timer. Table 2.5 gives a summary of the features of the timers.

Timer type	Timer	Counter resolution	Counter type	Prescaler factor	DMA request generation	Capture/compare channels	Complementary outputs
Advanced control	TIM1, TIM8	16-bit	Up, down, Up/down	Any integer between 1 and 65536	Yes	4	3
General-purpose	TIM2, TIM5	32-bit	Up, down, Up/down	Any integer between 1 and 65536	Yes	4	No
General-purpose	TIM3, TIM4	16-bit	Up, down, Up/down	Any integer between 1 and 65536	Yes	4	No
General-purpose	TIM15	16-bit	Up	Any integer between 1 and 65536	Yes	2	1

Table 2.5: STM32L476xx timers © STMicroelectronics. Used with permission.

2.3.11 Interrupts

Interrupts are important features of all microcontrollers. In this section the interrupt mechanism of the STM32L476xx processor is described in greater detail.

An interrupt is an external or an internal (e.g. timer) event that requests the CPU to halt what it is currently executing, and immediately jump to execute a different code. The code that the CPU jumps to is known as the Interrupt Service Routine (ISR). Interrupts are asynchronous events since the CPU does not know when the interrupt will occur in a program as it can occur at any time. The time between the generation of an interrupt request and the entry into the ISR is known as the 'interrupt latency'. The faster the interrupt latency (i.e. the lower the value) is always better as this shows that the system response is fast. When an interrupt occurs, the CPU remembers the location of the next instruction it was going to execute by storing the program counter in a register or memory location (usually called a stack). The CPU returns to this instruction and resumes normal operation after completing the ISR code and returning from the interrupt. In addition to saving the program counter, the CPU may also store the important register values, such as the CPU status register, flags, and other registers of importance. Interrupt request that triggered the interrupt may also be disabled so that further interrupts from the same source are not recognized until the present one returns.

In a microcontroller, the CPU may be designed to respond to large number of external and internal interrupts, for example in the region of 10 to over 100. Each interrupt source usually, but not necessarily has a dedicated area of memory where the ISR code is expected

to be located at. In some processors, multiple interrupt sources share the same memory locations for their ISRs. It is then up to the user program to determine the actual source of the interrupt. Interrupts are always disabled when a processor is started. The following conditions must be met before an interrupt can be accepted by the CPU:

- interrupt from the required source must be enabled;
- the source must generate an interrupt request;
- processor global interrupts must be enabled.

External interrupts usually occur when a port pin is driven low (i.e. on the falling edge) or high (i.e. on the rising edge), or if there is change in the state of the port pin (low-to-high or high-to-low). Internal timer interrupts occur for example when a timer overflows or underflows.

Interrupt sources usually have priorities assigned to them. A higher priority interrupt can stop the execution of a lower priority interrupt and grab the CPU. When the higher priority ISR completes its processing, the lower priority ISR can resume.

It is important that the ISR code should consume as little processor time as possible. This is especially true when multiple interrupts are enabled in a program. If an ISR takes long time, then it may cause delayed response to other interrupts that may need fast response. Interrupts can be masked under software control so that requests from such sources cannot be accepted. Some processors also have non-maskable interrupt (NMI) sources that cannot be masked (or disabled) under software control.

Various registers inside the CPU must be configured before an external or an internal interrupt can be accepted by the CPU. The next sections describe the STM32F407 interrupt features and show the bit mappings of the registers involved in configuring various interrupt sources.

The STM32L476xx processor supports up to 240 IRQs and 1 NMI. There are 16 levels of priority with level zero the highest and level 15, the lowest.

The following conditions must be met at the device and inside the CPU for an interrupt to be accepted:

At the device

- Each interrupt source has a separate enable bit. The bit must be set for the interrupt to be accepted.
- Each interrupt source has a separate flag bit. Hardware sets this bit when it makes an interrupt request. This bit must be cleared in ISR by software.

In the CPU

- Interrupt request is received via the Nested Vectored Interrupt Controller (NVIC). The highest priority interrupt request is sent to the CPU.
- Global interrupt enable bit must be enabled in the PRIMASK register.
- The priority level of the requesting source must be higher than the base priority BASEPRI.

The NVIC has the following features:

- 82 maskable interrupt channels;
- 16 programmable priority levels;
- Low-latency interrupt handling.

STM32L476xx interrupt mechanism is designed for fast and efficient interrupt handling. The first line of a C code in the interrupt service routine is guaranteed to be executed after 12 cycles for a zero wait state memory system. In addition, the interrupt latency is fully deterministic so that the same interrupt latency is entered from any point in the code.

External interrupts

The STM32L476xx processor has 23 external interrupt sources, where 16 sources are available at the port pins, and the remaining ones are for events like RTC, Ethernet, USB and so on. External interrupt/event controller (EXTI) consists of up to 23 edge detectors for generating event/interrupt requests. Each input line can be independently configured to select the type and the corresponding trigger event (rising or falling, or both). Each line can also be masked independently. A pending register maintains the status line of the interrupt requests.

Figure 2.26 shows the EXTI block diagram. The edge of the external interrupt input is first detected. The interrupt source can be masked in software by configuring the interrupt mask register. After going through the pending request register, the interrupt is presented to the NVIC interrupt controller.

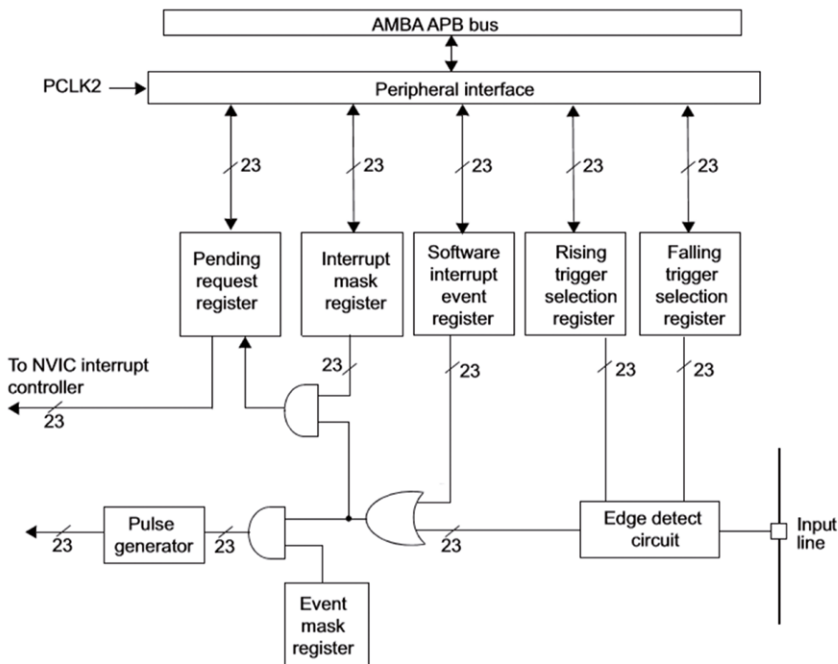


Figure 2.26: NVIC block diagram (© STMicroelectronics. Used with permission).

External interrupt GPIO mapping is shown in Figure 2.27. 16 multiplexers select GPIO pins as external interrupts EXTI0 to EXTI15. The multiplexer inputs are selected via 4-bit fields of $EXTICR[k]$ registers, where $k = 0$ through 3. Pin 0 of all the external interrupt GPIO pins share the line EXTI0, Pin 1 of all the external interrupt GPIO pins share the line EXTI1 and so on up to pin 15 of the GPIO pins which share the EXTI15 line.

Register SYSCFG_EXTICRx is a 16-bit register made up of 4 4-bit nibbles. As shown in Figure 2.28, nibble 0 is EXTI0, nibble 1 is EXTI1, nibble 2 is EXTI2 and nibble 3 is EXTI3. The interrupt lines are selected as follows: As an example, EXTIx = 0 selects PAX, 1 selects PBx, 2 selects PCx, etc. EXTICR1 selects EXTI3-EXTI0; EXTICR2 selects EXTI7-EXTI4, etc.

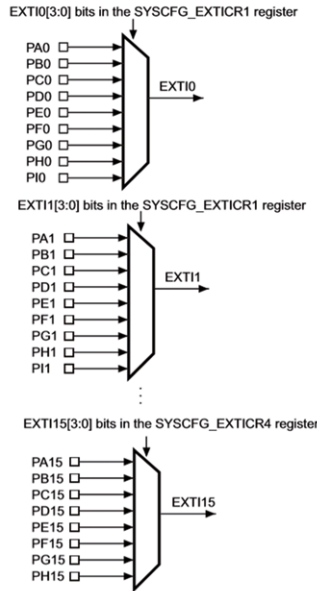


Figure 2.27: External interrupt GPIO mapping
(© STMiroelectronics. Used with permission).

SYSCFG_EXTICR1			
EXTI3	EXTI2	EXTI1	EXTI0

Figure 2.28: Register SYSCFG_EXTICR1.

Table 2.6 to Table 2.9 show the register SYSCFG_EXTICx and corresponding interrupt pin numbers.

EXTI3 (bits 15...12)	EXTI2 (bits 11...8)	EXTI1 (bits 7...4)	EXTI0 (bits 3...0)
0: PA3	0: PA2	0: PA1	0: PA0
1: PB3	1: PB2	1: PB1	1: PB0
2: PC3	2: PC2	2: PC1	2: PC0
3: PD3	3: PD2	3: PD1	3: PD0
4: PE3	4: PE2	4: PE1	4: PE0
5: PF3	5: PF2	5: PF1	5: PF0

Table 2.6: SYSCFG_EXTICR1 (© STMiroelectronics. Used with permission).

EXTI7 (bits 15...12)	EXTI6 (bits 11...8)	EXTI5 (bits 7...4)	EXTI4 (bits 3...0)
0: PA7	0: PA6	0: PA5	0: PA4
1: PB7	1: PB6	1: PB5	1: PB4
2: PC7	2: PC6	2: PC5	2: PC4
3: PD7	3: PD6	3: PD5	3: PD4
4: PE7	4: PE6	4: PE5	4: PE4
5: PF7	5: PF6	5: PF5	5: PF4

Table 2.7: SYSCFG_EXTICR2 (© STMicroelectronics. Used with permission).

EXTI11 (bits 15...12)	EXTI10 (bits 11...8)	EXTI9 (bits 7...4)	EXTI8 (bits 3...0)
0: PA11	0: PA10	0: PA9	0: PA8
1: PB11	1: PB10	1: PB9	1: PB8
2: PC11	2: PC10	2: PC9	2: PC8
3: PD11	3: PD10	3: PD9	3: PD8
4: PE11	4: PE10	4: PE9	4: PE8
5: PF11	5: PF10	5: PF9	5: PF8

Table 2.8: SYSCFG_EXTICR3 (© STMicroelectronics. Used with permission).

EXTI15 (bits 15...12)	EXTI14 (bits 11...8)	EXTI13 (bits 7...4)	EXTI12 (bits 3...0)
0: PA15	0: PA14	0: PA13	0: PA12
1: PB15	1: PB14	1: PB13	1: PB12
2: PC15	2: PC14	2: PC13	2: PC12
3: PD15	3: PD14	3: PD13	3: PD12
4: PE15	4: PE14	4: PE13	4: PE12
5: PF15	5: PF14	5: PF13	5: PF12

Table 2.9: SYSCFG_EXTICR4 (© STMicroelectronics. Used with permission).

The other EXTI lines are connected as follows:

- EXTI line 16 is connected to the PVD output;
- EXTI line 17 is connected to the RTC Alarm event;
- EXTI line 18 is connected to the USB OTG FS Wakeup event;
- EXTI line 19 is connected to the Ethernet Wakeup event;
- EXTI line 20 is connected to the USB OTG HS (configured in FS) Wakeup event;
- EXTI line 21 is connected to the RTC Tamper and TimeStamp events;
- EXTI line 22 is connected to the RTC Wakeup event.

EXTI registers

The following registers can be configured for external interrupts.

Interrupt Mask Register (EXTI_IMR): Used to mask an interrupt line. A 0 masks the interrupt, while a 1 unmask the interrupt line. The register configuration is shown in Figure 2.29.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved									MR22	MR21	MR20	MR19	MR18	MR17	MR16
									r/w	r/w	r/w	r/w	r/w	r/w	r/w
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MR15	MR14	MR13	MR12	MR11	MR10	MR9	MR8	MR7	MR6	MR5	MR4	MR3	MR2	MR1	MR0
r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w

Bits 31:23 Reserved, must be kept at reset value.

Bits 22:0 **MRx**: Interrupt mask on line x

0: Interrupt request from line x is masked

1: Interrupt request from line x is not masked

Figure 2.29: EXTI_IMR register (©STMicroelectronics. Used with permission).

Rising Trigger Selection Register (EXTI_RTSR): Used to set the interrupt source as rising-edge (low-to-high). A 1 sets the corresponding interrupt line as rising. The register configuration is shown in Figure 2.30.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved									TR22	TR21	TR20	TR19	TR18	TR17	TR16
									r/w	r/w	r/w	r/w	r/w	r/w	r/w
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TR15	TR14	TR13	TR12	TR11	TR10	TR9	TR8	TR7	TR6	TR5	TR4	TR3	TR2	TR1	TR0
r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w

Bits 31:23 Reserved, must be kept at reset value.

Bits 22:0 **TRx**: Rising trigger event configuration bit of line x

0: Rising trigger disabled (for Event and Interrupt) for input line

1: Rising trigger enabled (for Event and Interrupt) for input line

Figure 2.30: EXTI_RTSR register (© STMicroelectronics. Used with permission).

Falling Trigger Selection Register (EXTI_FTSR): Used to set the interrupt source as falling (high-to-low). A 1 sets the corresponding interrupt line as falling. The register configuration is shown in Figure 2.31.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved									TR22	TR21	TR20	TR19	TR18	TR17	TR16
									r/w	r/w	r/w	r/w	r/w	r/w	r/w
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TR15	TR14	TR13	TR12	TR11	TR10	TR9	TR8	TR7	TR6	TR5	TR4	TR3	TR2	TR1	TR0
r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w

Bits 31:23 Reserved, must be kept at reset value.

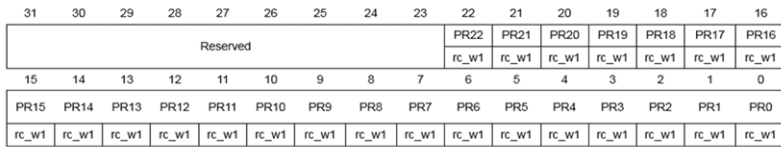
Bits 22:0 **TRx**: Falling trigger event configuration bit of line x

0: Falling trigger disabled (for Event and Interrupt) for input line

1: Falling trigger enabled (for Event and Interrupt) for input line.

Figure 2.31: EXTI_FTSR register (© STMicroelectronics. Used with permission).

Pending Register (EXTI_PR): A bit in the register is set when the selected edge event arrives on an interrupt line. A 1 must be written by the ISR to clear the pending state of the interrupt (i.e. to cancel the IRQ request). The register configuration is shown in Figure 2.32.



Bits 31:23 Reserved, must be kept at reset value.

Bits 22:0 **PRx**: Pending bit

0: No trigger request occurred

1: selected trigger request occurred

This bit is set when the selected edge event arrives on the external interrupt line.

This bit is cleared by programming it to '1'.

Figure 2.32: EXTI_PR register (© STMicroelectronics. Used with permission).

The positions and the priorities of external interrupt lines are shown in Table 2.10. All pins with same number are connected to line with same number. They are multiplexed to one line. You cannot use two pins on one line simultaneously. Notice that EXTI5_9 is shared and handles interrupts from pins 5 to 9. Similarly, EXTI10_15 is shared and handles interrupts from pins 10 to 15.

IRQ no	Priority	Interrupt Line	Description
6	13	EXTI0	EXTI Line0 interrupt
7	14	EXTI1	EXTI Line0 interrupt
8	15	EXTI2	EXTI Line0 interrupt
9	16	EXTI3	EXTI Line0 interrupt
10	17	EXTI4	EXTI Line0 interrupt
23	30	EXTI9_5	EXTI Line 9-5 interrupts
40	47	EXTI15_10	EXTI Line 15-10 interrupts

Table 2.10: External interrupt positions and priorities.

2.4 Summary

In this Chapter the basic features of the Cortex-M processors have been described. In addition, the architecture of the STM32L467xx microcontroller has been described in some detail since this is the microcontroller that is used in the Nucleo-L476RG development board, and this board is the 'hardware platform' for all the projects in this book.

In the next Chapters we will be looking at the software tools that can be used in the development of STM32 processor-based projects.

CHAPTER 3 • STM32 Nucleo Software Development Tools (Toolchains)

3.1 Overview

This Chapter is about the software development tools (Integrated Development Environments, IDEs, or Toolchains) that can be used to develop software for the Nucleo boards. Brief specifications of the popular Nucleo development tools are discussed.

3.2 Integrated development environments supporting the Nucleo boards

The Nucleo boards are supported by several software development tools in the form of Integrated Development Environments (IDEs). In general, most IDEs provide a built-in text editor, compiler, debugger, simulator, and a program upload tool so that the developed program code can be uploaded to the program memory of the target microcontroller. The compiler is usually a version of C/C++.

Some of the popular IDEs supporting the Nucleo boards are:

- Embedded Workbench for Arm (EWARM) by IAR Systems
- Arm Mbed
- MDK-ARM by Keil
- TrueSTUDIO by Atollic
- System Workbench for STM32 (SW4STM32) by AC6
- STM32CubeIDE

Brief details of these IDEs are given in the following sections.

3.3 Embedded Workbench for Arm (EWARM)

This is a professional compiler developed by the IAR Systems. The compiler is based on a highly optimizing C/C++ and it supports devices based on Arm Cortex-M0, Cortex-M0+, Cortex-M3, Cortex-M4, and Cortex-M7. A debugger with simulator is supported by the EWARM.

Two free evaluation versions of the software are available: time limited version, and code size limited version. The differences between the two versions are:

30-day time-limited version

- License will expire after 30 days
- Source code for runtime libraries is not included
- No support for MISRA C
- C-RUN is size limited to 12 KB of code, excluding constant data
- Limited technical support
- Must not be used for commercial product development

Size-limited version

- Code size limited to 32 KB
- Source code for runtime libraries is not included
- No support for MISRA C
- C-RUN is not available
- Limited technical support

When you start the product the first time you will be asked to register to get the evaluation license. The choice between the time limited version and the code size limited versions is made after the product is installed.

3.3.1 Installing the EWARM

The steps to install the EWARM are as follows:

- Download the IDE from the following link. At the time of writing this book the latest version was 8.50. This IDE occupies about 1.45 GB and has the name EWARM-CD-8505-26295.exe (Figure 3.1):

<https://www.iar.com/iar-embedded-workbench/tools-for-arm/arm-cortex-m-edition/>

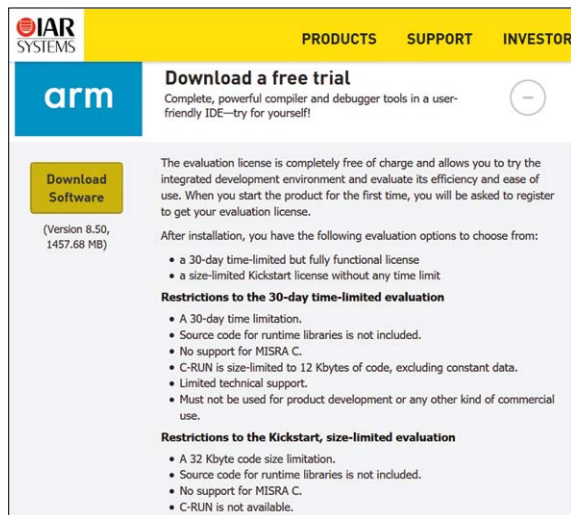


Figure 3.1: Installing the EWARM.

- Copy the downloaded file to a folder and then double click on it to start the installation process. You should see the screen as in Figure 3.2 at the beginning of the installation. Click on the first option to start downloading. The installation process may take several minutes and you should wait until it is finished.



Figure 3.2: Beginning of the installation.

- Register with IAR Systems to get an evaluation license (see Figure 3.3). You will have to confirm your registration by clicking at the link sent to your registration address. Enter the license number into the given form and you should then be able to use the product.

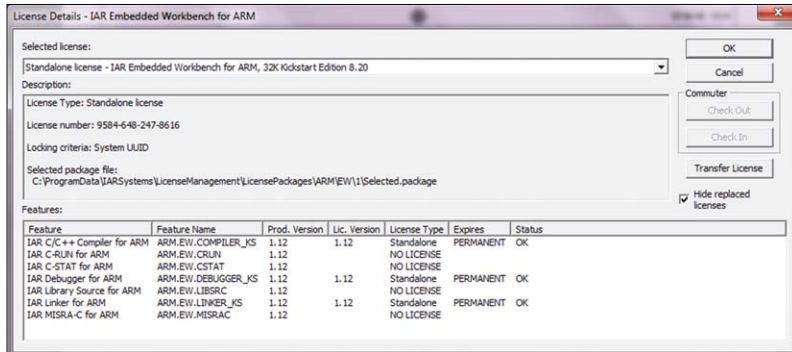


Figure 3.3: Registering with IAR Systems.

- Figure 3.4 shows the startup screen of the EWARM.

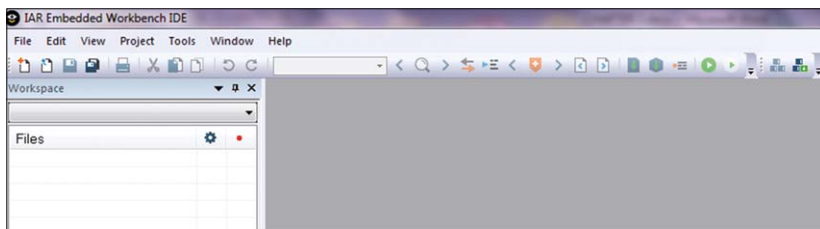


Figure 3.4: EWARM startup screen.

3.4 Arm Mbed

Mbed is a free online Arm compiler that can be used over an Internet link. It is a platform and operating system based on 32-bit Arm Cortex-M microcontrollers. Mbed is supported by over 60 partners and a community of 200,000 developers. Mbed is a free online IDE consisting of an online code editor and a compiler. Only a web browser is required to access Mbed and develop Arm based programs. Programs are compiled on the Cloud using the Arm C/C++ compiler. Developing a project using Mbed is very easy since the user needs to pick a supported development board, write the application program, and then upload the program to the board.

The Mbed compiler is easy to use and it supports a large number of Arm processors and software libraries. Using Mbed the compiled code can easily be uploaded to the program memory of the target Arm microcontroller as a simple copy operation. First time Mbed users must register before they can use the Mbed. The steps to register and import the Nucleo-L476RG board to the Mbed working environment are given below.

- Register to Mbed through the following link. Figure 3.5 shows the Mbed registration/login screen. Click **Signup** at the right-hand side and enter your details: <https://os.mbed.com/account/login/>

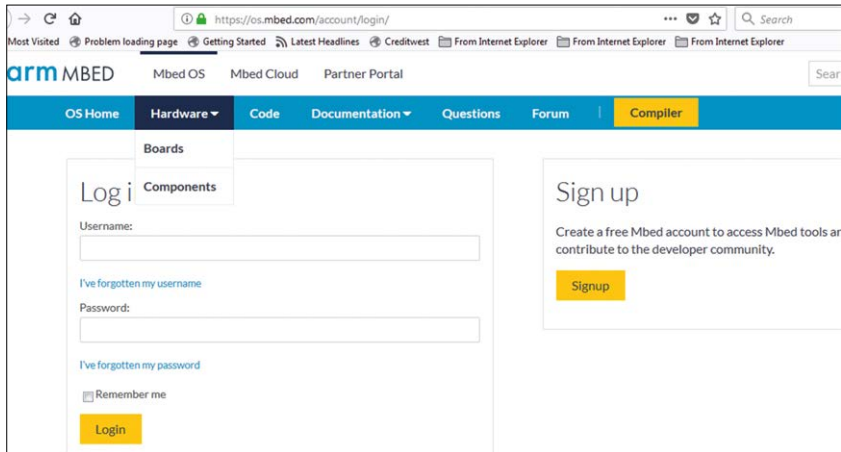


Figure 3.5: Mbed registration/login screen.

- Click **Hardware** → **Boards** and then click on picture Nucleo-L476RG (see Figure 3.6).

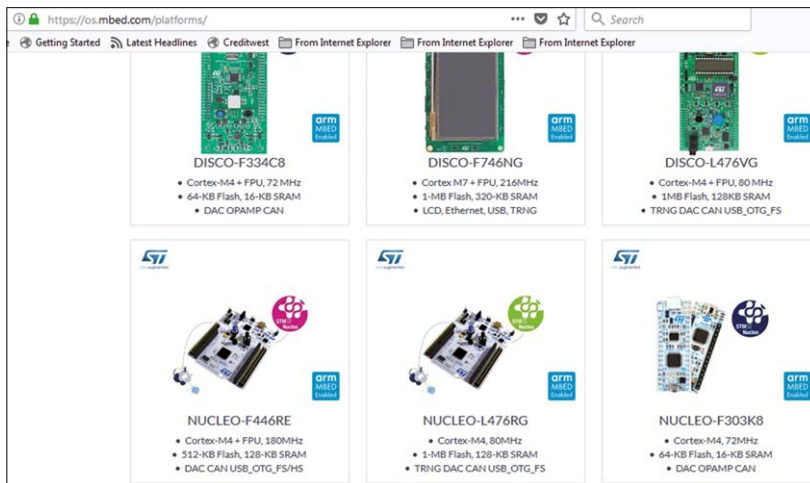


Figure 3.6: Click on picture Nucleo-L476RG.

- Click on **Add to your Mbed Compiler** to add the board to your compiler environment (see Figure 3.7).



Figure 3.7: Add the Nucleo-L476RG board to your compiler environment.

- Click **Compiler** to start the compiler (see Figure 3.8).

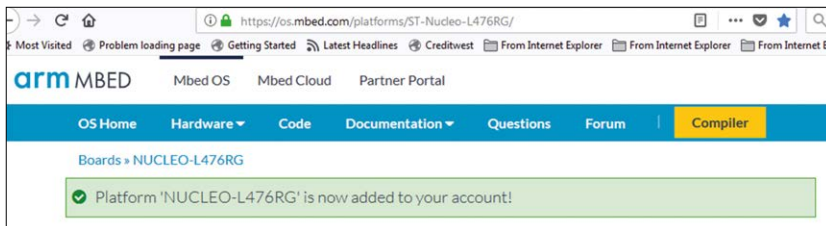


Figure 3.8: Starting the Mbed compiler.

3.5 MDK-ARM

MDK-ARM is a professional IDE for the Arm processors, developed by Keil. A code size limited Lite version of this IDE, called the MDK-Lite, is available free of charge, intended for product evaluation, small projects, and for the educational market.

The Lite version does not compile, assemble or link program code which is greater than 32 KB. Also, the debugger supports programs that are 32 KB or smaller, and the compiler does not generate a disassembly listing of the machine code generated.

The steps to download and install the Lite version of this IDE are as follows:

- Download the IDE from the following link (see Figure 3.9). You should download the MDK-Core. You will have to register before you can download the IDE.

<http://www2.keil.com/mdk5/editions/lite>

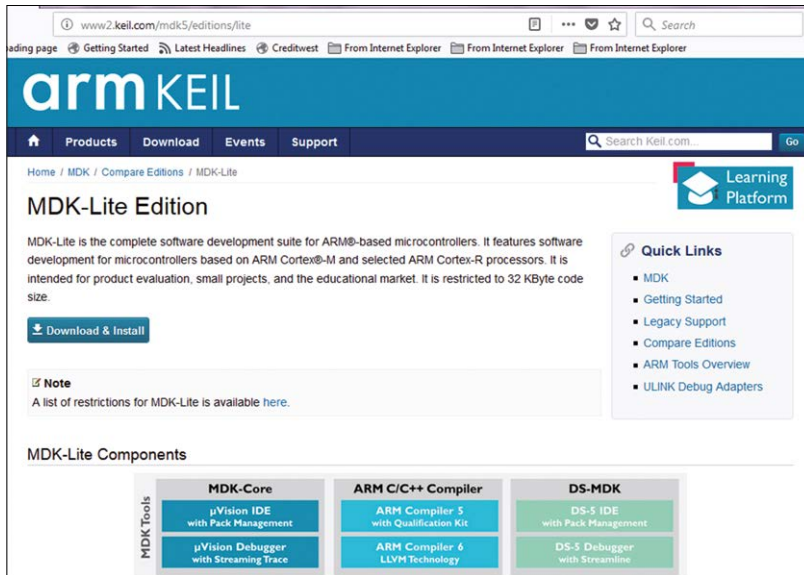


Figure 3.9: Download the MDK-Lite IDE.

- Save the file in a folder on your computer.
- Double click on the filename to install the IDE.

Figure 3.10 shows the startup screen of the MDK-Lite IDE.

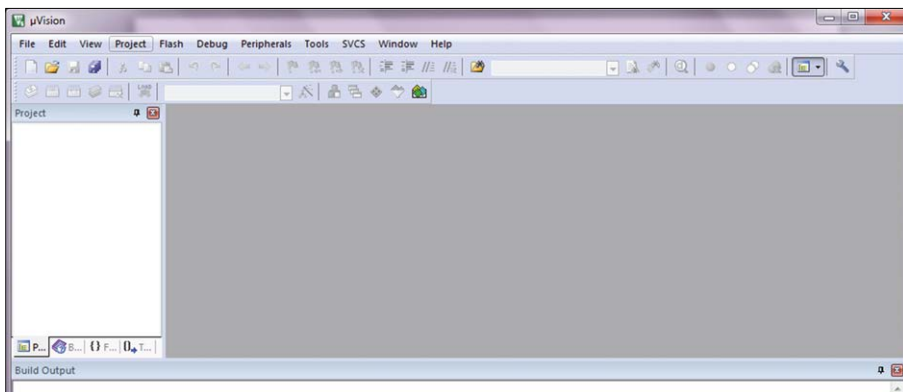


Figure 3.10: MDK-Lite startup screen.

3.6 TrueSTUDIO

TrueSTUDIO is a professional IDE for the development of Arm based projects. This IDE has a Lite version that supports the STM32 Arm processors. The steps to download and install the IDE are given below.

- Download the IDE from the following site (see Figure 3.11) by selecting the latest version.

<https://atollic.com/truestudio/>



Figure 3.11: Download the TrueSTUDIO Lite.

Notice that the TrueSTUDIO has been replaced with the STM32CubeIDE and will not be discussed further in this book.

3.7 System Workbench for STM32 (SW4STM32)

The System Workbench toolchain, called SW4STM32, is a free development environment based on Eclipse, which supports the full range of STM32 microcontrollers and associated boards. The System Workbench toolchain and its collaborative website have been built by AC6, a service company providing training and consultancy on embedded systems.

The key features of the SW4STM32 are:

- Support for STM32 microcontrollers, STM32 Nucleo boards, Discovery kits and Evaluation boards, as well as STM32 firmware (Standard Peripheral library or STM32Cube HAL).
- Free of charge and no code size limit.
- GCC C/C++ compiler and GDB based debugger.
- Eclipse IDE with team-work management, compatible with Eclipse plug-ins.
- ST-LINK support.

The steps to download and install the SW4STM32 are given below.

- Go to the following link and register:
<https://www.openstm32.org/System%2BWorkbench%2Bfor%2BSTM32>
- Click on the link **Installing System Workbench for STM32 with Installer**.
- Click **register** to register at the site by giving your details.
- Login to the site.
- Click on **Installing System Workbench for STM32 with installer**.
- Click on **Downloading the System Workbench for STM32 installer** (see Figure 3.12).

Downloading the product installer

System Workbench for STM32 can be installed with the executable product installer. Before installation, please ensure that the user account has the administrative access right or that you possess the administrator password.

You should first download the offline installer from [Downloading the System Workbench for STM32 installer](#).

Installing on Windows

1. Click and launch *System Workbench for STM32* installer.
2. Wait until the installer window is displayed.
3. The first page describes the product features. Click on the "Next" button.
4. Please read and accept the license agreement to continue the installation. Click on the "Next" button.
5. Choose the installation path (default: C:\Ac6\SystemWorkbench). Please avoid choosing path with space such as "C:\Program Files...". Click on the "Next" button.
6. A warning message is displayed :
 1. If the directory does not exist, it proposes to create the installation directory.
 2. If the directory exists, the installer will suppress the installation folder. (Installing on top of older might causes several issues)
7. The next page shows the list of packs that will be installed.
8. Select if you want to create Start menu and desktop shortcuts.
9. The following page shows on the installation settings. Click on the "Next" button to proceed installation.
10. Wait until the installation is done and click on the "Quit" button when "Finish" message is displayed on the installation progress.

Figure 3.12: Downloading the SW4STM32 installer.

- Select the type of system you have (e.g. Windows 10 64 bit) as in Figure 3.13. At the time of writing this book the latest version v2.9.

Windows 7 & 10

The Windows version is available for 32 and 64 bit systems. Note that we will need to install a device driver to communicate with the ST-Link debug. you **must** select the installer that fits your system. Installing the 32 bit version on a 64 bit Windows system will **not** work. If you have problems downloading the executable file (.exe), try downloading and extracting the ZIP file. In both cases you are advised to also download the MD5 or SHA256 checksum to verify the integrity of your download.

- Latest Windows 64 bit installer Version v2.9, updated on Friday, April 12, 2019 at 16:41:04 CEST):
 - Installer: [install_sw4stm32_win_64bits-v2.9.exe](#)
 - MD5 sum d9aa87c2c98003aca0d6ac21cc89f84c in [install_sw4stm32_win_64bits-v2.9.exe.md5](#)
 - SHA256 sum 000cc9745b22e82ac51ecd0b880a56c8ee60cd512d6ed85a6dd401f3fd67f22 in [install_sw4stm32_win_64bits-v2.9.exe.sha256](#)
 - Zipped installer: [install_sw4stm32_win_64bits-v2.9.zip](#)
 - MD5 sum cd9e6fedca96332a2237a40ceca4b1c9 in [install_sw4stm32_win_64bits-v2.9.zip.md5](#)
 - SHA256 sum 4aa855564ae9a8d1882647b9fea11c2de1ddb6230389a6db9e08a93028b5282b in [install_sw4stm32_win_64bits-v2.9.zip.sha256](#)

Figure 3.13: Select the type of system you have (part of the display is shown).

- Copy the downloaded file in to a folder. The author installed the software on a Windows 10 computer with 64-bit MCU. At the time of writing this book the filename was **install_sw4stm32_win_64bits-v2.9.exe**
- Double click on the file to install the SW4STM32.
- Figure 3.14 shows the SW4STM32 startup screen.

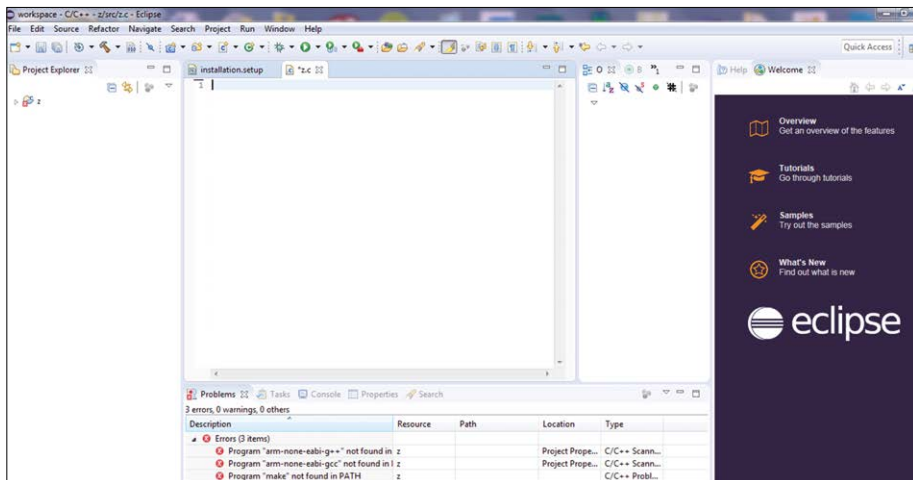


Figure 3.14: SW4STM32 startup screen.

3.8 STM32CubeIDE

STM32CubeIDE is an advanced C/C++ development platform with peripheral configuration, code generation, code compilation, and debug features for STM32 microcontrollers and microprocessors. Based on the Eclipse/CDT framework and GCC toolchain for the development, it allows the integration of the hundreds of existing plugins that complete the features of the Eclipse IDE.

STM32CubeIDE integrates STM32 configuration and project creation functionalities from STM32CubeMX to offer all-in-one tool experience and save installation and development time. After the selection of an empty STM32 MCU or MPU, or preconfigured microcontroller or microprocessor from the selection of a board or the selection of an example, the project is created and initialization code generated. At any time during the development, the user can return to the initialization and configuration of the peripherals or middleware and regenerate the initialization code with no impact on the user code.

STM32CubeIDE also includes standard and advanced debugging features including views of CPU core registers, memories, and peripheral registers, as well as live variable watch, Serial Wire Viewer interface, or fault analyzer.

The projects in this book have been developed using the STM32CubeIDE software package. This software package is a very powerful Integrated Development Environment, having the following features:

- based on Eclipse/CDT, with support of Eclipse add-ons, GNU C/C++ for Arm toolchain and GDB debugger;
- integration of services from STM32CubeMX, including: STM32 microcontroller, development platform and example project selection, pinout, clock, peripheral, and middleware configuration, project creation and generation of the initialization code, enhanced STM32Cube Expansion Packages;
- advanced debug features, including CPU core, peripheral register, and memory views, live variable watch view, system analysis and real-time tracing, CPU fault analysis tool;

- support of ST-LINK and J-Link debug probes;
- import projects from Atollic TrueSTUDIO and AC6 System Workbench for STM32;
- multi-OS support, including Windows, Linux, macOS (64-bit versions only).

The steps to install the STM32CubeIDE are given below.

- Go to the following STMicroelectronics website (Figure 3.15):
<https://www.st.com/en/development-tools/stm32cubeprog.html>

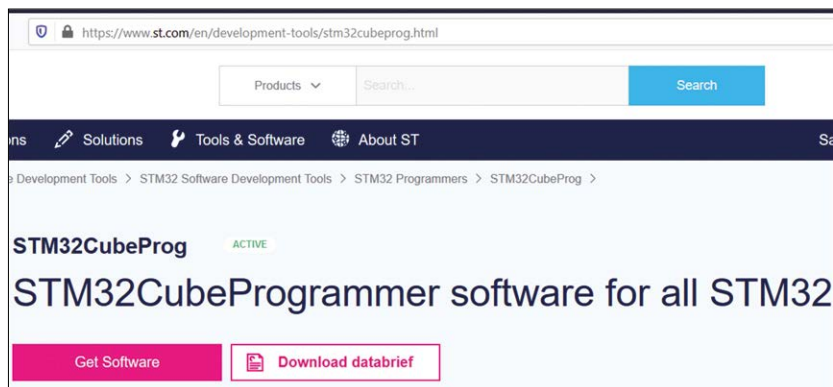


Figure 3.15: Click to install the software.

- Click **Get Software**. You may have to enter your details and then accept the licence conditions (see Figure 3.16).

Get Software

If you have an account on my.st.com, login and download the software without any further validation steps.

[Login/Register](#)

If you don't want to login now, you can download the software by simply providing your name and e-mail address in the form below and validating it.

This allows us to stay in contact and inform you about updates of this software.

For subsequent downloads this step will not be required for most of our software.

First Name:

Last Name:

E-mail address:

☐ I have read and understood the [Sales Terms & Conditions](#), [Terms of Use](#) and [Privacy Policy](#)

Figure 3.16: Entering your details.

- Validate your details by clicking the link sent to your email address.
- Download the software. At the time of writing this book the software was named: **en.stm32cubeprog_v2-5-0.zip**.

- Unzip, and install the software on your PC. A shortcut will be placed on your Desktop to activate the application.

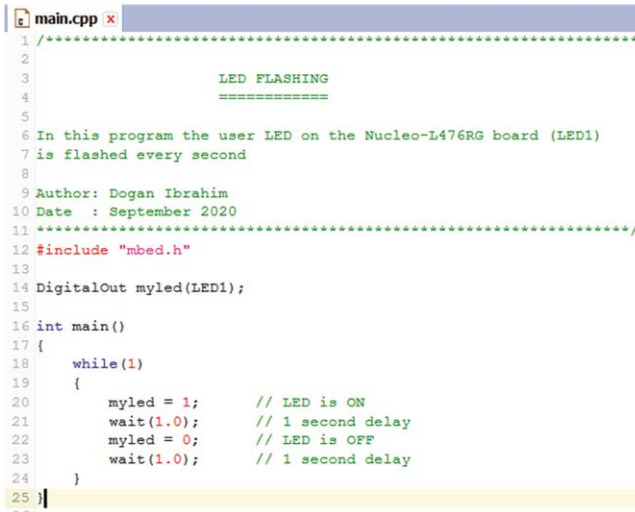
3.9 Summary

In this Chapter we have briefly seen how to download and install the popular STM32 Nucleo board software development tools. All the tools given in this Chapter are in the form of Integrated Development Systems which include a text editor, compiler and linker, debugger, simulator, and a program upload tool.

All the projects in this book have been developed using the STM32CubeIDE software package. For the sake of completeness, in the next Chapter we shall be looking at a simple example and show how the Mbed can be used to create a simple project.



- Click **OK**.
- Click on **main.cpp** at the working space (middle part) of the screen to open the program listing.
- Since we want the LED to flash every second, change the program as shown in Figure 4.4. Notice that we have also added comments to the program.



```

1  /*****
2
3          LED FLASHING
4          *****/
5
6  In this program the user LED on the Nucleo-L476RG board (LED1)
7  is flashed every second
8
9  Author: Dogan Ibrahim
10 Date : September 2020
11 *****/
12 #include "mbed.h"
13
14 DigitalOut myled(LED1);
15
16 int main()
17 {
18     while(1)
19     {
20         myled = 1;    // LED is ON
21         wait(1.0);    // 1 second delay
22         myled = 0;    // LED is OFF
23         wait(1.0);    // 1 second delay
24     }
25 }

```

Figure 4.4: Modified program to flash the LED every second.

- Click **Compile** to compile the program. If the program has compiled successfully you should see the message 'Success!' as in Figure 4.5. You should also see a pop-up window (Figure 4.6) asking you to save the binary file. Click **Save File** and **OK** to save the binary file in the Downloads folder of your PC.

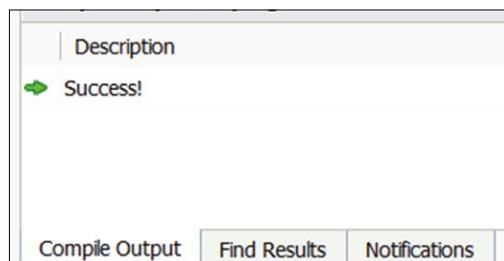


Figure 4.5: Successful programming.

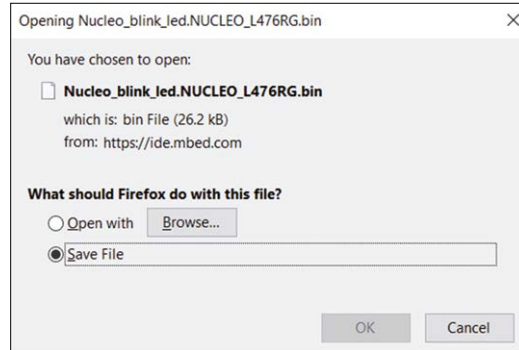


Figure 4.6: Save the binary file.

- You should now be ready to upload the created binary file to the program memory of your Nucleo-L476RG board. Connect the board to your PC through a suitable USB connector. You should see 'NODE_L476RG' listed as a device on your PC (Figure 4.7).

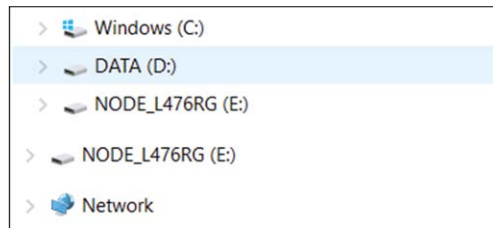


Figure 4.7: NODE_L476RG device.

- The saved binary file is given the default name **Nucleo_blink_led_NUCLEO_L476RG.bin**. Click and drag this binary file in the Downloads folder (Figure 4.8) and drop it on the NODE_L476 device. You should see the communications LED on the Nucleo-L476RG board flashing red and green as the program is uploaded to your Nucleo board. Wait until the upload is complete which is indicated by the LED turning to steady green colour.

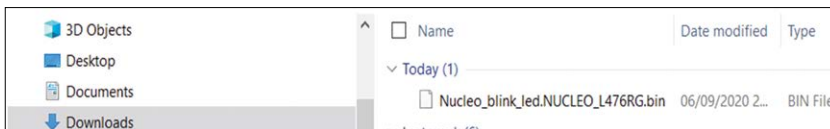


Figure 4.8: The compiled file in Download folder.

- You should now see the user LED flashing every second on your Nucleo-L476RG board.

It is worthwhile now to look at the operation of the program in some detail. The comments at the beginning of the program start with the `'/*'` character pair and terminate with the `'*/'` character pair. File **mbed.h** is included at the beginning of the program so that the

various commands used in the program can be recognized by the C compiler. Most of the pins on our board can be used as inputs or outputs and we have to configure the pin that will be used. This is done using a statement in the following format:

```
Type name(pin);
```

Where ‘Type’ indicates the function of the pin, such as **DigitalOut** or **DigitalIn** and so on. ‘name’ can be any name chosen by the programmer to identify the pin, and ‘pin’ refers to the pin’s actual name like in Figure 4.1 or Figure 4.2. In this example, pin LED1 is assigned the name **myled** and it is configured as an output using the statement:

```
DigitalOut myled(LED1);
```

The remainder of the program is executed in an endless loop formed using a **while(1)** statement. Inside this loop the LED is turned ON (`myled = 1`), one second delay is inserted, then the LED is turned OFF (`myled = 0`). This loop repeats after one second delay.

Note that LED1 is the reserved name for the on-board User LED. We can access any GPIO pin by specifying the port name and the pin number. For example, pin 1 of Port C can be accessed as: `PC_1`. This port pin can for example can be defined as an output pin using the statement:

```
DigitalOut mypin(PC_1);
```

Where **mypin** refers to pin 1 of Port C.

4.3 Summary

In this Chapter we have seen how to use the Mbed development environment in a simple project using the Nucleo-L476RG board. The program development process got described very briefly.

In next Chapter we shall be developing various projects using the STM32CubeIDE integrated development environment with the Nucleo-L476RG board.

CHAPTER 5 • STM32CubeIDE Nucleo-L476 Projects

5.1 Overview

In last chapter we have seen how Mbed can be used to create a very simple project. In this chapter we will be developing various projects using the STM32CubeIDE with the Nucleo-L476RG development board. Notice that all the project given in this book have been fully tested and are working.

The following will be specified and given for each project (where appropriate):

- project title;
- project description;
- aim of the project;
- project block diagram;
- project circuit diagram;
- project construction;
- project program listing;
- description of the program;
- additional work (optional).

The operation of some of the more complex projects will be described using the Program Description Language (PDL). This is a free-format, English-like description of the operation of the project using keywords such as BEGIN, END, IF, THEN, ELSE, WHILE, REPEAT, DO, DO FOREVER, and so on.

5.1.1 STM32cubeIDE GPIO library

The GPIO code generated by the STM32CubeIDE software is in the HAL library. The HAL library contains many functions that can be used to develop applications using the STM32 processors. The HAL library offers high-level, feature-oriented APIs that can be employed in the STM32CubeIDE toolchain to develop STM32 based projects. In this section we shall be looking briefly at the HAL GPIO library functions.

The HAL library is initialized with the statement: **HAL_Init()**. The following GPIO functions are offered.

A GPIO port is configured using a structure in the following format. In this example, GPIOA pin 0 is configured as an output:

```
GPIO_InitStruct.Pin = GPIO_PIN_0;
GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
GPIO_InitStruct.Speed = GPIO_SPEED_LOW;
HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
```

We can combine the pins as in the following example:

```
GPIO_InitStruct.Pin = GPIO_PIN_0 | GPIO_PIN_1 | GPIO_PIN_2;
```

or,

```
GPIO_InitStruct.Pin = GPIO_PIN_All;
```

The GPIO ports are configured in function called **MX_GPIO_Init** which is called automatically from the main program when the program is created.

The parameter **mode** can take the following values:

GPIO_MODE_INPUT	Input Floating Mode
GPIO_MODE_OUTPUT_PP	Output Push Pull Mode
GPIO_MODE_OUTPUT_OD	Output Open Drain Mode
GPIO_MODE_AF_PP	Alternate Function Push Pull Mode
GPIO_MODE_AF_OD	Alternate Function Open Drain Mode
GPIO_MODE_ANALOG	Analogue Mode
GPIO_MODE_IT_RISING	Ext. Int. with Rising edge detection
GPIO_MODE_IT_FALLING	Ext. Int. with Falling edge detection
GPIO_MODE_IT_RISING_FALLING	Ext. Int. with Rising/Falling edge detection
GPIO_MODE_EVT_RISING	Ext. Event Mode with Rising edge detection
GPIO_MODE_EVT_FALLING	Ext. Event Mode with Falling edge detection
GPIO_MODE_EVT_RISING_FALLING	Ext. Event Mode with Rising/Falling detection

The parameter **speed** can take the following values (refer to the datasheet):

GPIO_SPEED_FREQ_LOW	IO works at 2 MHz
GPIO_SPEED_FREQ_MEDIUM	12.5 MHz to 50 MHz
GPIO_SPEED_FREQ_HIGH	25 MHz to 100 MHz
GPIO_SPEED_FREQ_VERY_HIGH	50 MHz to 200 MHz

The pullup/pull down can be:

GPIO_NOPULL	No Pull-up or Pull-down activation
GPIO_PULLUP	Pull-up activation
GPIO_PULLDOWN	Pull-down activation

HAL_GPIO_WritePin: This function is used to set (logic 1) or reset (logic 0) a GPIO pin. In the following example, pin 1 of GPIOA is set to logic 1:

```
HAL_GPIO_WritePin(GPIOA, GPIO_PIN_1, GPIO_PIN_SET);
```

Similarly, the same pin is reset to logic 0 using the following statement:

```
HAL_GPIO_WritePin(GPIOA, GPIO_PIN_1, GPIO_PIN_RESET);
```

HAL_GPIO_ReadPin: This function is used to read the state of a GPIO pin. In the following example, the state of pin 1 of GPIOA is read and stored in a variable called **data**:

```
data = HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_1);
```

HAL_GPIO_TogglePin: This function is used to toggle the state of a pin. In the following example the state of pin 8 of GPIOA is toggled:

```
HAL_TogglePin(GPIOA, GPIO_PIN_8);
```

HAL_GPIO_LockPin: This function is used to lock a port pin. The configuration of the locked pin cannot be changed until the next reset.

HAL_GPIO_EXTI_IRQHandler: This function handles EXTI interrupt requests.

HAL_GPIO_EXTI_Callback: This function implements the callback.

In addition, there are functions to implement the GPIO Alternate Function selections. We can also use the following commands for GPIO functions:

GPIOx -> ODR: This command is used to send data to a complete 16-bit port. In the following example, all bits of GPIOA are set to logic 1:

```
GPIOA -> ODR = 0xFFFF; or GPIOA->ODR=0b1111111111111111;
```

To set a single bit in a port, e.g. to set bit 8 of GPIOA without affecting the other bits we can use the following statement:

```
GPIOA -> ODR |= GPIO_PIN_8;
```

Similarly, to clear for example pin 8 of GPIOA without affecting other bits we can write:

```
GPIOA -> ODR &= ~GPIO_PIN_8;
```

We can alternatively use the BSRRL/BSRRH command to set/reset a bit or a group of bits. The least significant 16 bits are used to atomically set pin values to VDD whereas the most significant 16 bits are used to atomically clear pin values to GND. In the following example, bit 8 of GPIOA is set without affecting the other bits:

```
GPIOA -> BSRRL = GPIO_PIN_8;
```

Similarly, to clear bit 8 of GPIOA without affecting other bits we can write:

```
GPIOA -> BSRRH = GPIO_PIN_8;
```

GPIOx -> IDR: This command is used to read the state of a complete 16-bit port. In the following example, all bits of GPIOA are read and stored in a variable called **data**:

```
data = GPIOA->IDR;
```

To read the state of an individual pin, for example, to read the state of GPIOA pin 13 and store in variable **data** we can write:

```
data = GPIOA -> IDR & GPIO_PIN_13;
```

In the remaining sections of this Chapter we will be developing various projects. **It is important to be familiar with the first project steps as this project describes in detail how to create a program using the STM32CubeIDE, how to compile the program, and how to upload the executable code to the Nucleo-L476RG development board. In this book, the GPIO port pins are defined as either Pn_x or as Pnx where 'n' is the port name (A, B, C, D etc) and 'x' is the pin number (0, 1, 2, 3, 4, etc). For example, PC_0 and PC0 are both the same port.**

5.2 Project 1: Lighthouse flashing LED

Description

In this project the user LED on the Nucleo-L476RG board is used for simplicity. The LED is flashed in a group of two quick flashes every second. The flashing rate is assumed to be 200 ms. This type of flashing is identified as **GpFI(2)** in maritime lighthouse lights. Thus, the required flashing sequence can be expressed as follows:

```
LED ON
Wait 200 ms
LED OFF
Wait 100 ms
LED ON
Wait 200 ms
LED OFF
Wait 100 ms
Wait 400 ms
```

The aim

The aim of this project is to show how an LED can be turned on and off at different rates. The project shows the steps to create a program, to compile it, and then to upload the executable code to the Nucleo development board.

Block diagram

Figure 5.1 shows the block diagram of the project where the user LED on the board is used in this project.

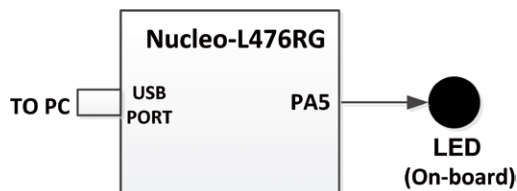


Figure 5.1: Block diagram of the project.

Circuit diagram

Figure 5.2 shows the user LED connected to port pin PA5 of the board. A 510-ohm current limiting resistor is used on the board to limit the LED current.

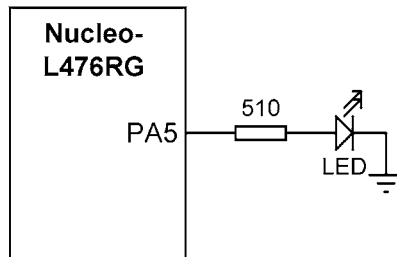


Figure 5.2: Circuit diagram of the project.

The steps to create, compile, and upload the program

- Start the STM32CubeIDE program by double clicking on its icon.
- STM32CubeIDE uses a working directory. Assuming the program has been installed to the default disk and folder, this directory is:
C:\Users\<user name>\STM32CubeIDE\<workspace number>

The workspace number starts from **workspace_1.3.3** by default.

- Figure 5.3 shows the workspace screen. Click **Launch** to accept it.

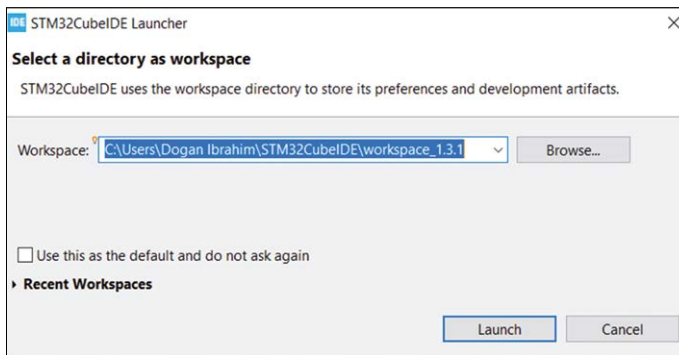


Figure 5.3: Workspace screen.

- Click **Start new STM32 project** (see Figure 5.4)



Figure 5.4: Click to start a new project.

- You will now be presented with the **Target Selection** screen.
- Enter **STM32L476RG** in textbox **Part Number Search** and then click on STM32L476RG to select the MCU as shown in Figure 5.5. You will also see brief specifications of the selected MCU at the top middle part of the screen.

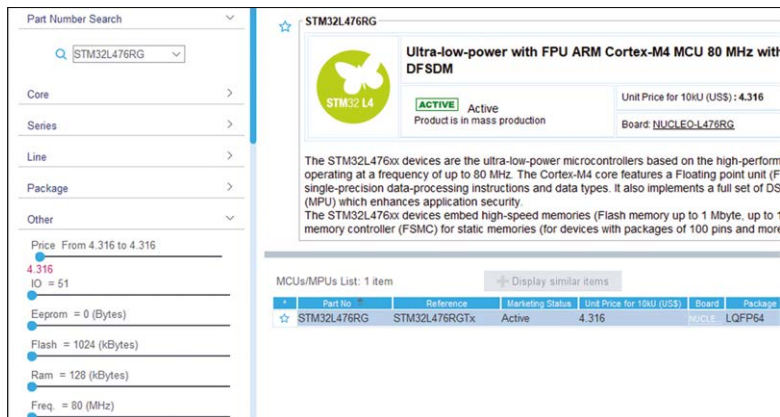


Figure 5.5: Select the MCU.

- Click **Next**
- Assign a name to your project. e.g. **FlashLED** as shown in Figure 5.6.

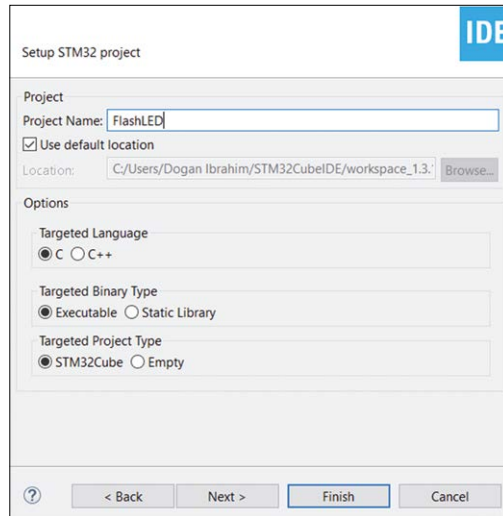


Figure 5.6: Naming your project.

- Click **Finish**.
- Click **Yes** to accept the STM32CubeMX perspective. You will now be presented with the STM32 chip layout where we have to select and configure the pins that will be used in the project.
- Enter **PA5** at the bottom right hand of the screen as shown in Figure 5.7 to locate pin PA5 (this is the port pin where the on-board user LED is connected to). You should see pin PA5 on the chip layout flashing.

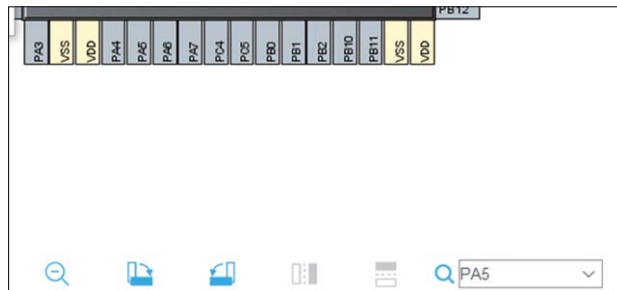


Figure 5.7: Locate pin PA5.

- Click on pin PA5 and select **GPIO_Output** to configure the pin as an output (Figure 5.8). The pin colour should turn to green.

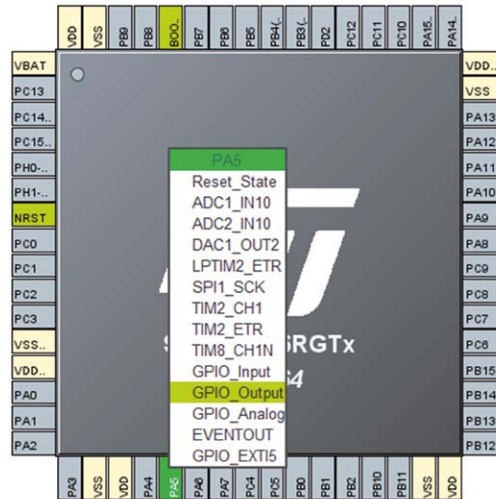


Figure 5.8: Configuring PA5 as an output.

- We now have to configure the clock for the MCU. Click tab **Clock Configuration**. You should see the clock configuration screen like in Figure 5.9.

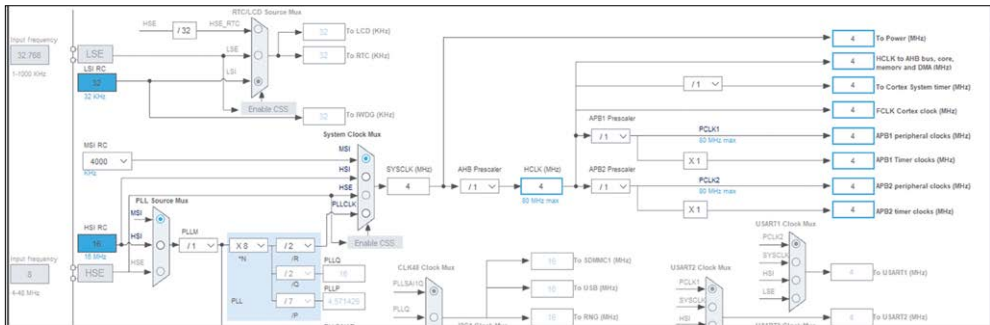


Figure 5.9: Clock configuration screen (part of the screen is shown).

- Here, the required clock option can be chosen by selecting the multiplexer inputs and pre-scaler values. In this project we will be using the MCU's 16 MHz internal clock. Let's see how to set the system clock and the peripheral clocks to operate at the maximum speed of 80 MHz. In Figure 5.10 the internal 16 MHz clock is selected (shown as dark colour) by selecting **HSI** at the **PLL Source Mux**. **PLLM** is set to 1 and the **PLL multiplier** is selected as 10, this making the clock as $16 \times 10 = 160$ MHz. The clock is then divided by 2 via the **/R** selector to give 80 MHz. The **System Clock Mux** is configured to select the **PLLCLK** which is now 80 MHz. **AHB Prescaler**, **APB1 Prescaler** and **APB2 Prescaler** are set to 1 so that the system clock and the peripheral clocks are now configured to operate at 80 MHz. Clocks to other parts of the processor (e.g. USART, ADC, LPTIMx, I2Cx etc can be configured similarly).



- ```

61=/**
62 * @brief The application entry point.
63 * @retval int
64 */
65=int main(void)
66 {
67 /* USER CODE BEGIN 1 */
68
69 /* USER CODE END 1 */
70
71 /* MCU Configuration-----
72
73 * Reset of all peripherals, Initializes the Flash interface
74 HAL_Init();

```

- You will see various empty code sections with template headings, such as **USER CODE BEGIN 1, USER CODE END 1**, etc.
- Move down to section starting with statements **static void MX\_GPIO\_Init(void)** which is the procedure where the I/O ports are initialized. You should see the following code which configures port pin **PA5** as digital output:

● 80

- We can now enter our program code to flash the LED as required. Go to the beginning of the program and enter the following statement:

```
#define myled GPIO_PIN_5
```

- Go to the main program loop and enter the following statements. The LED is turned ON when the argument is set to **GPIO\_PIN\_SET**, and turned OFF when the argument is **GPIO\_PIN\_RESET**. Function **HAL\_Delay(n)** creates **n** milliseconds of delay in the program:

```
while (1)
{
 HAL_GPIO_WritePin(GPIOA, myled, GPIO_PIN_SET);
 HAL_Delay(200);
 HAL_GPIO_WritePin(GPIOA, myled, GPIO_PIN_RESET);
 HAL_Delay(100);
 HAL_GPIO_WritePin(GPIOA, myled, GPIO_PIN_SET);
 HAL_Delay(200);
 HAL_GPIO_WritePin(GPIOA, myled, GPIO_PIN_RESET);
 HAL_Delay(100);
 HAL_Delay(400);
}
```

- We are now ready to compile the program. First, let us set the compiler to generate **Release** code instead of Debug code. This is done by clicking **Project**, followed by **Build Configurations**, and then **Set Active**, and set to **Release**.
- Click **Project** followed by **Build All** to compile the program.
- Look at the bottom of the screen and make sure that there are no errors (see Figure 5.12).

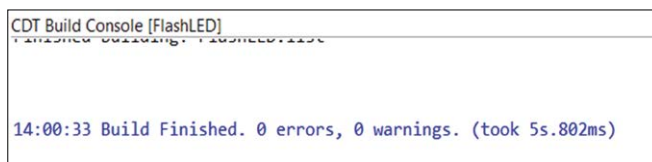


Figure 5.12: Checking if there are errors.

- If the program has been compiled without any errors, then the executable binary file will be located in the working directory (see Figure 5.13):  
C:\Users\<user name>\STM32CubeIDE\workspace\_1.3.1\FlashLED\Release\FlashLED.bin

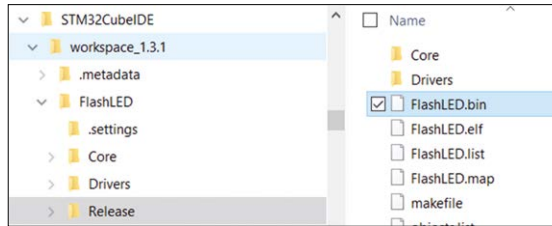


Figure 5.13: Location of the compiled binary file.

- Now, make sure the development board is plugged into the USB port of your PC. Drag the binary file and drop it to drive **NODE\_L476RG** as shown in Figure 5.14. You should see the communication LED flashing while the executable file is uploaded to the MCU of the development board.

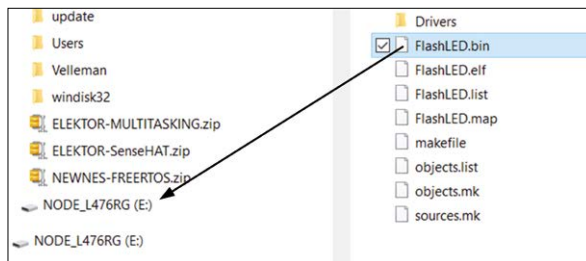


Figure 5.14: Drag and drop the binary file.

You should now see the user LED flashing as required.

Figure 5.15 shows the complete program listing of the project. Notice that most of the comments have been removed to make the program more readable.

```
/* USER CODE BEGIN Header */
/**
 * @file : main.c
 * @brief : Main program body
 *
 * @attention
 *
 * <h2><center>© Copyright (c) 2020 STMicroelectronics.
 * All rights reserved.</center></h2>
 *
 * This software component is licensed by ST under BSD 3-Clause license,
 * the «License»; You may not use this file except in compliance with the
 * License. You may obtain a copy of the License at:
 *
 * opensource.org/licenses/BSD-3-Clause
 *
 */
```

```

#include «main.h»
#define myled GPIO_PIN_5

/* Private function prototypes -----*/
void SystemClock_Config(void);
static void MX_GPIO_Init(void);

int main(void)
{
 /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
 HAL_Init();

 /* Configure the system clock */
 SystemClock_Config();

 /* Initialize all configured peripherals */
 MX_GPIO_Init();

 /* USER CODE BEGIN WHILE */
 while (1)
 {
 HAL_GPIO_WritePin(GPIOA, myled, GPIO_PIN_SET);
 HAL_Delay(200);
 HAL_GPIO_WritePin(GPIOA, myled, GPIO_PIN_RESET);
 HAL_Delay(100);
 HAL_GPIO_WritePin(GPIOA, myled, GPIO_PIN_SET);
 HAL_Delay(200);
 HAL_GPIO_WritePin(GPIOA, myled, GPIO_PIN_RESET);
 HAL_Delay(100);
 HAL_Delay(400);
 }
}

void SystemClock_Config(void)
{
 RCC_OscInitTypeDef RCC_OscInitStruct = {0};
 RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

 RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
 RCC_OscInitStruct.HSISState = RCC_HSI_ON;
 RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
 RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
 RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
 RCC_OscInitStruct.PLL.PLLM = 1;
 RCC_OscInitStruct.PLL.PLLN = 10;
 RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV7;

```

```
RCC_OscInitStruct.PLL.PLLQ = RCC_PLLQ_DIV2;
RCC_OscInitStruct.PLL.PLLR = RCC_PLLR_DIV2;
if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
{
 Error_Handler();
}

RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
 |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_4) != HAL_OK)
{
 Error_Handler();
}

if (HAL_PWREx_ControlVoltageScaling(PWR_REGULATOR_VOLTAGE_SCALE1) != HAL_OK)
{
 Error_Handler();
}
}

static void MX_GPIO_Init(void)
{
 GPIO_InitTypeDef GPIO_InitStruct = {0};

 /* GPIO Ports Clock Enable */
 __HAL_RCC_GPIOA_CLK_ENABLE();

 /*Configure GPIO pin Output Level */
 HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_RESET);

 /*Configure GPIO pin : PA5 */
 GPIO_InitStruct.Pin = GPIO_PIN_5;
 GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
 GPIO_InitStruct.Pull = GPIO_NOPULL;
 GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
 HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
}

void Error_Handler(void)
{
}
}
```

```

#ifdef USE_FULL_ASSERT

void assert_failed(uint8_t *file, uint32_t line)
{
}

#endif

/***** (C) COPYRIGHT STMicroelectronics *****/

```

*Figure 5.15: Program: FlashLED.*

Some noticeable aspects of the program are given below.

Inside the main program, functions **HAL\_Init()** is called to initialize the HAL library. Additionally, function **SystemClock\_Config()** is called to configure the MCU clock, and function **MX\_GPIO\_Init()** is called to configure the GPIO ports:

```

HAL_Init();

/* Configure the system clock */
SystemClock_Config();

/* Initialize all configured peripherals */
MX_GPIO_Init();

```

The MCU clock is configured with the function **SystemClock\_Config**:

```

RCC_OscInitTypeDef RCC_OscInitStruct = {0};
RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
RCC_OscInitStruct.HSIState = RCC_HSI_ON;
RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
RCC_OscInitStruct.PLL.PLLM = 1;
RCC_OscInitStruct.PLL.PLLN = 10;
RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV7;
RCC_OscInitStruct.PLL.PLLQ = RCC_PLLQ_DIV2;
RCC_OscInitStruct.PLL.PLLR = RCC_PLLR_DIV2;

```

The function configures the MCU clock to use the internal oscillator (HSI) with the PLLM set to 1, and PLLN set to 10, resulting in 80 MHz.



The AHB, APB1 and APB2 bus clocks are configured in the remaining parts of the function:

```
RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
 |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;
```

Notice that a port pin can easily be toggled using the following statement:

```
HAL_GPIO_TogglePin(pin name);
```

### Using an external crystal for timing

In this project we have used the internal clock of the MCU. There are time-critical applications where we may want to use an accurate external crystal for timing. Nucleo-L476RG development board includes an 8 MHz external crystal connected to the clock inputs of the MCU. The steps to use the external crystal for 80 MHz operation are given below.

- In the configuration screen, click **System Core** and select **RCC**. Then select **Crystal/Ceramic Resonator** for the **High Speed Clock (HSE)**, as shown in Figure 5.16

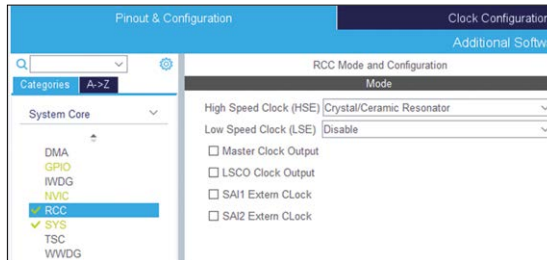


Figure 5.16: Select external clock (HSE).

- Click tab **Clock Configuration** to configure the external clock for 80MHz operation (see Figure 5.17).
- Select **Input frequency** as 8 MHz.
- Select **HSE** at the **PLL Source Mux**.
- Set **PLLM** to 1, **N** to X20, and **/R** to 2.
- Select **PLLCLK** at the **System Clock Mux**.
- You should see that 80 MHz clock got set.

- Click **File**, then **Save**, and **YES** to generate code.

```
void SystemClock_Config(void)
{
 RCC_OscInitTypeDef RCC_OscInitStruct = {0};
 RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

 /** Initializes the CPU, AHB and APB busses clocks
 */
 RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSE;
 RCC_OscInitStruct.HSEState = RCC_HSE_ON;
 RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
 RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSE;
 RCC_OscInitStruct.PLL.PLLM = 1;
 RCC_OscInitStruct.PLL.PLLN = 20;
 RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV7;
 RCC_OscInitStruct.PLL.PLLQ = RCC_PLLQ_DIV2;
 RCC_OscInitStruct.PLL.PLLR = RCC_PLLR_DIV2;
 if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
 {
 Error_Handler();
 }

 /** Initializes the CPU, AHB and APB busses clocks
 */
 RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
 |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
 RCC_ClkInitStruct.SYSClkSource = RCC_SYSCCLKSOURCE_PLLCLK;
 RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCCLK_DIV1;
 RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
 RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

 if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_4) != HAL_OK)
 {

```

```

 Error_Handler();
}

if (HAL_PWREx_ControlVoltageScaling(PWR_REGULATOR_VOLTAGE_SCALE1) != HAL_OK)
{
 Error_Handler();
}
}

```

### Saving the program

The program can be saved by clicking **File**, followed by **Save As**. Selecting **Src** saves the main program (**main.c**) in the specified directory (Figure 5.18).

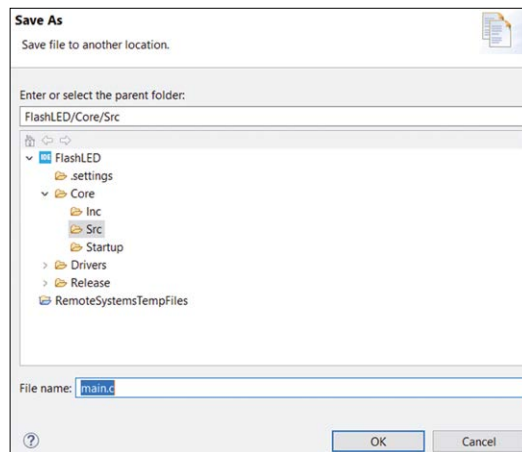


Figure 5.18: Saving the main program.

It is preferable to save all components of a project by exporting the project. The steps are (Figure 5.19):

- click **File**, followed by **Export**;
- select **File System** and click **Next**;
- click **Select All**;
- click **Browse** to select the folder where the project will be saved to;
- click **Finish** to save the project.

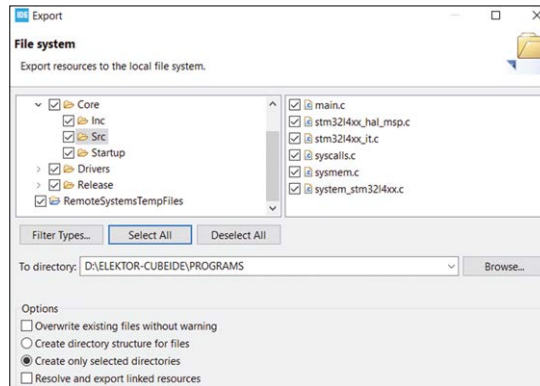


Figure 5.19: Saving the project.

### 5.3 Project 2: Alternately Flashing LEDs

#### Description

In this project two LEDs are connected to GPIO ports PA6 and PA7 of the Nucleo-L476RG board (see Figure 5.1 and Figure 5.2). The LEDs flash alternately every 500 ms.

#### The aim

The aim of this project is to show how external components (e.g. LEDs) can be connected to the Nucleo-L476RG board.

#### Block diagram

Figure 5.20 shows the block diagram of the project.

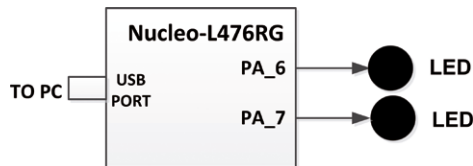


Figure 5.20: Block diagram of the project.

#### Circuit diagram

Figure 5.21 shows the circuit diagram of the project where two LEDs are connected to port pins PA6 (pin 13 on morpho connector CN10) and PA7 (pin 15 on morpho connector CN10) through 470-ohm current limiting resistors.

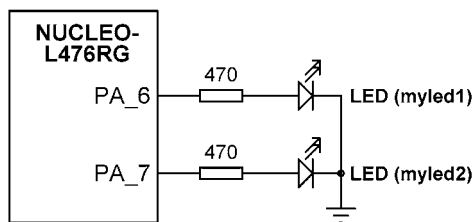


Figure 5.21: Circuit diagram of the project.

### The construction

As pictured in Figure 5.22, the project was built on a breadboard where jumper wires were used to make connections to the connector on the Nucleo board. Pin 8 on connector CN7 was used as the ground pin.

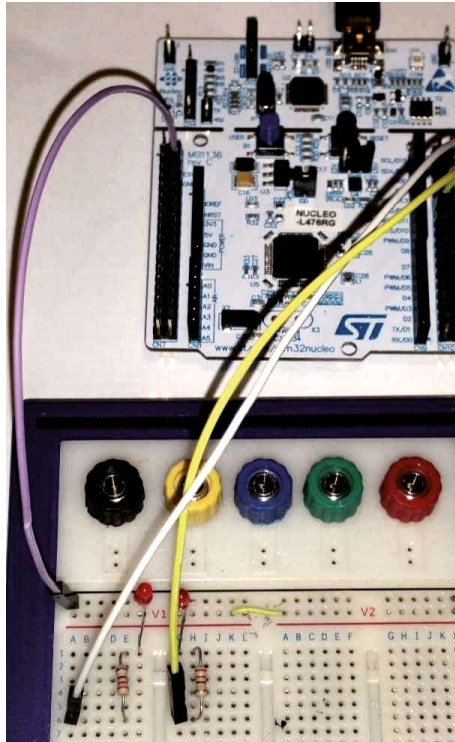


Figure 5.22: Project constructed on a breadboard.

### Program listing

The steps to create the program are given below.

- Start the STM32CubeIDE.
- You will be presented with the screen asking you whether or not to accept the existing workspace. Click **Launch** to accept the default workspace (you can also create a new workspace if you wish).
- Click **File**, followed by **New** and then select **STM32 Project**.
- Select the MCU type STM32L476RG as described in the previous project.
- Click **Next** and Give a name to the project, e.g. **AlternateLEDs** (see Figure 5.23), and click **Finish**.

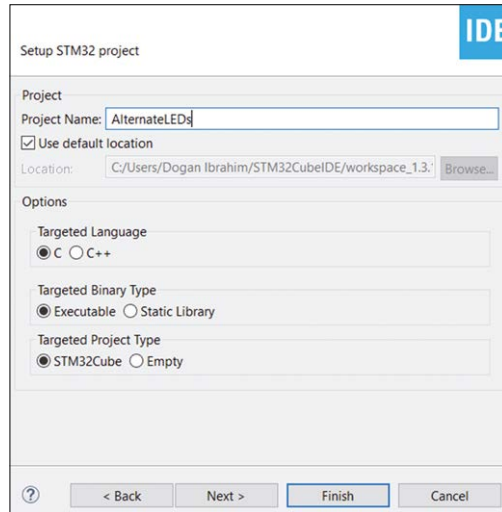


Figure 5.23: Assigning a name to the project.

- Click **YES** to configure the pins.
- Configure pins **PA6** and **PA7** as digital outputs (**GPIO\_Output**) as described in the previous project.
- Click tab **Clock Configuration** and set the MCU clock to 80 MHz as described in the previous project.
- Click **File**, followed by **Save** and click **YES** to generate code.
- Click **Core**, followed by **Src** and double click **main.c** to display the main program.

We can now enter our code. Enter the following two lines under the statement **#include "stm32l4xx\_hal.h"**

```
#define myled1 GPIO_PIN_6
#define myled2 GPIO_PIN_7
```

Enter the following code inside the 'while' loop to flash the two LEDs alternately:

```
while (1)
{
 HAL_GPIO_WritePin(GPIOA,myled1,GPIO_PIN_SET);
 HAL_GPIO_WritePin(GPIOA,myled2,GPIO_PIN_RESET);
 HAL_Delay(1000);
 HAL_GPIO_WritePin(GPIOA,myled1,GPIO_PIN_RESET);
 HAL_GPIO_WritePin(GPIOA,myled2,GPIO_PIN_SET);
 HAL_Delay(1000);
}
```

Make sure that the compiler is set in **Release** mode, and click **Project**, followed by **Build All**. The program will compile after a few seconds. You should make sure that there are no errors. Drag the program binary file (**AlternateLEDs.bin**) in working directory, under folder **AlternateLEDs\Release** (Figure 5.24) to device **NODE\_L476RG** as in the previous project. You should see the two LEDs flashing alternately.

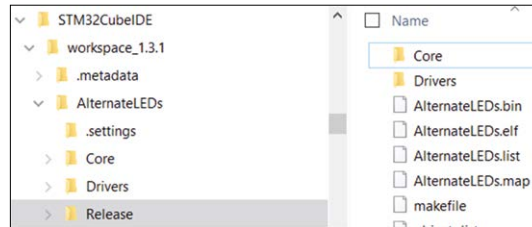


Figure 5.24: Program binary file.

Figure 5.25 shows the complete program listing (comments have been removed for clarity).

```
/* USER CODE BEGIN Header */
/**
 * *****
 * @file : main.c
 * @brief : Main program body
 * *****
 * @attention
 *
 * <h2><center>© Copyright (c) 2020 STMicroelectronics.
 * All rights reserved.</center></h2>
 *
 * This software component is licensed by ST under BSD 3-Clause license,
 * the «License»; You may not use this file except in compliance with the
 * License. You may obtain a copy of the License at:
 *
 * opensource.org/licenses/BSD-3-Clause
 *
 * *****
 */
/* USER CODE END Header */

/* Includes -----*/
#include «main.h»
#define myled1 GPIO_PIN_6
#define myled2 GPIO_PIN_7

/* Private function prototypes -----*/
void SystemClock_Config(void);
static void MX_GPIO_Init(void);
```

```

int main(void)
{
 /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
 HAL_Init();

 /* Configure the system clock */
 SystemClock_Config();

 /* Initialize all configured peripherals */
 MX_GPIO_Init();

 /* USER CODE BEGIN WHILE */
 while (1)
 {
 HAL_GPIO_WritePin(GPIOA,myled1,GPIO_PIN_SET);
 HAL_GPIO_WritePin(GPIOA,myled2,GPIO_PIN_RESET);
 HAL_Delay(1000);
 HAL_GPIO_WritePin(GPIOA,myled1,GPIO_PIN_RESET);
 HAL_GPIO_WritePin(GPIOA,myled2,GPIO_PIN_SET);
 HAL_Delay(1000);
 }
}

void SystemClock_Config(void)
{
 RCC_OscInitTypeDef RCC_OscInitStruct = {0};
 RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

 RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
 RCC_OscInitStruct.HSISState = RCC_HSI_ON;
 RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
 RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
 RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
 RCC_OscInitStruct.PLL.PLLM = 2;
 RCC_OscInitStruct.PLL.PLLN = 20;
 RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV7;
 RCC_OscInitStruct.PLL.PLLQ = RCC_PLLQ_DIV2;
 RCC_OscInitStruct.PLL.PLLR = RCC_PLLR_DIV2;
 if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
 {
 Error_Handler();
 }

 RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLOCK
 |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
 RCC_ClkInitStruct.SYSClockSource = RCC_SYSCLOCKSOURCE_PLLCLK;
 RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLOCK_DIV1;

```



```

RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_4) != HAL_OK)
{
 Error_Handler();
}

if (HAL_PWREx_ControlVoltageScaling(PWR_REGULATOR_VOLTAGE_SCALE1) != HAL_OK)
{
 Error_Handler();
}
}

static void MX_GPIO_Init(void)
{
 GPIO_InitTypeDef GPIO_InitStruct = {0};

 /* GPIO Ports Clock Enable */
 __HAL_RCC_GPIOA_CLK_ENABLE();

 /*Configure GPIO pin Output Level */
 HAL_GPIO_WritePin(GPIOA, GPIO_PIN_6|GPIO_PIN_7, GPIO_PIN_RESET);

 /*Configure GPIO pins : PA6 PA7 */
 GPIO_InitStruct.Pin = GPIO_PIN_6|GPIO_PIN_7;
 GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
 GPIO_InitStruct.Pull = GPIO_NOPULL;
 GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
 HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
}

void Error_Handler(void)
{
}

#ifdef USE_FULL_ASSERT

void assert_failed(uint8_t *file, uint32_t line)
{
}

#endif

/***** (C) COPYRIGHT STMicroelectronics *****/

```

*Figure 5.25 Full program: AlternateLEDs.*

### 5.4 Project 3: 'Moving' LEDs

#### Description

In this project eight LEDs are connected to GPIO ports PC0 to PC7 of the Nucleo-L476RG board (see Figure 5.1 and Figure 5.2) where PC0 is the least significant bit and PC7 is the most significant bit. The LEDs turn ON/OFF in a moving manner where only one LED is ON at any given time as shown in Figure 5.26. A 250-ms delay is inserted between each output.

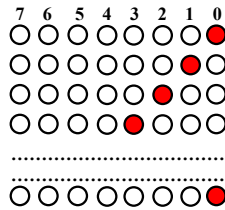


Figure 5.26: 'Moving' LEDs.

A 250-ms delay is inserted between each output.

#### The aim

The aim of this project is to show how many external LEDs can be connected to the Nucleo-L476RG board.

#### Block diagram

Figure 5.27 shows the block diagram of the project.

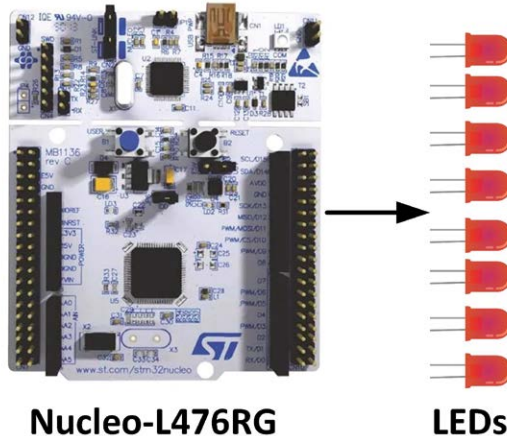


Figure 5.27: Block diagram of the project.

### Circuit diagram

Figure 5.28 shows the circuit diagram of the project where four LEDs are connected to port pins PC0 to PC7 through 470-ohm current limiting resistors as shown below:

| Port Pin | CN7 Header | CN10 Header |
|----------|------------|-------------|
| PC_0     | 38         | -           |
| PC_1     | 36         | -           |
| PC_2     | 35         | -           |
| PC_3     | 37         | -           |
| PC_4     | -          | 34          |
| PC_5     | -          | 6           |
| PC_6     | -          | 4           |
| PC_7     | -          | 19          |

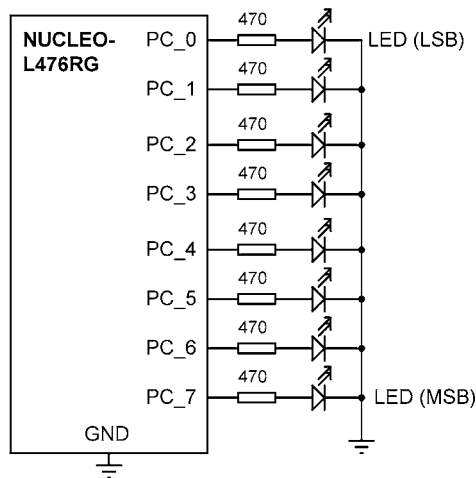


Figure 5.28: Circuit diagram of the project.

### The construction

As shown in Figure 5.29, the project was built on a breadboard where jumper wires were used to make connections between the four LEDs and the connectors on the Nucleo-L476RG board. Pin 8 on connector CN7 was used as the ground pin.

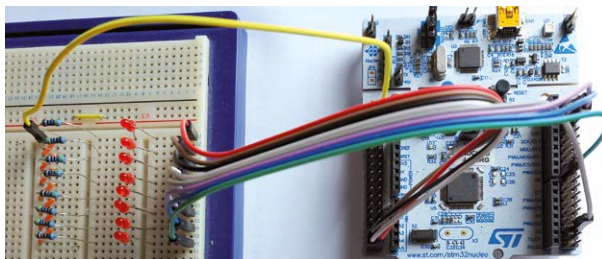


Figure 5.29: Project constructed on a breadboard.

## Program listing

The steps to create the program are given below.

- Start the STM32CubeIDE.
- Create a program with the name **MovingLEDs** as in the previous projects.
- Select the MCU type STM32L476RG as described in the previous project.
- Configure pins PC0, PC1, PC2, PC3, PC4, PC5, PC6 and PC7 as digital outputs (**GPIO\_Output**) as described in the previous projects.
- Click tab **Clock Configuration** and set the MCU clock to 80 MHz as described in the previous projects.
- Click **File**, followed by **Save** and click **YES** to generate code.
- Click **Core**, followed by **Src** and double click **main.c** to display the main program.
- We can now enter our code. Enter the following lines under the statement **#include "stm32l4xx\_hal.h"**. led0 to led7 define the LED connections and array LEDS stores these connections.

```
#define led0 GPIO_PIN_0
#define led1 GPIO_PIN_1
#define led2 GPIO_PIN_2
#define led3 GPIO_PIN_3
#define led4 GPIO_PIN_4
#define led5 GPIO_PIN_5
#define led6 GPIO_PIN_6
#define led7 GPIO_PIN_7
int LEDS[] = {led0, led1, led2, led3, led4, led5, led6, led7};
```

- Inside the main program, the code to move the LEDs is as follows. A loop is formed which iterates 8 times to control each LED individually. A 250-ms delay is inserted between each output:

```
while (1)
{
 for(j = 0; j <= 7; j++)
 {
 HAL_GPIO_WritePin(GPIOC,LEDS[j],GPIO_PIN_SET);
 HAL_Delay(250);
 HAL_GPIO_WritePin(GPIOC,LEDS[j],GPIO_PIN_RESET);
 HAL_Delay(250);
 }
}
```

- Set the IDE to generate **Release** code (Click **Project**, followed by **Build Configurations**, and then **Set Active**, and select **Release**), and click **Project** followed by **Build All** to build the program. Make sure that there are no errors.

- Drag and drop the binary file **MovingLEDs.bin** in subdirectory **MovingLEDs\Release** of the working directory to device **NODE\_L476RG**. You should see the LEDs flashing one after the other as shown in Figure 5.26.

Figure 5.30 shows the program listing with most of the comments removed for clarity. It is interesting to notice that the GPIO ports are initialized by calling function **MX\_GPIO\_Init()** which includes the following statements:

```
static void MX_GPIO_Init(void)
{
 GPIO_InitTypeDef GPIO_InitStruct = {0};

 /* GPIO Ports Clock Enable */
 __HAL_RCC_GPIOC_CLK_ENABLE();

 /*Configure GPIO pin Output Level */
 HAL_GPIO_WritePin(GPIOC, GPIO_PIN_0|GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3
 |GPIO_PIN_4|GPIO_PIN_5|GPIO_PIN_6|GPIO_PIN_7,
GPIO_PIN_RESET);

 /*Configure GPIO pins : PC0 PC1 PC2 PC3
 PC4 PC5 PC6 PC7 */
 GPIO_InitStruct.Pin = GPIO_PIN_0|GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3
 |GPIO_PIN_4|GPIO_PIN_5|GPIO_PIN_6|GPIO_PIN_7;
 GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
 GPIO_InitStruct.Pull = GPIO_NOPULL;
 GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
 HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);
}

/* USER CODE BEGIN Header */
/**
 * *****
 * @file : main.c
 * @brief : Main program body
 * *****
 * @attention
 *
 * <h2><center>© Copyright (c) 2020 STMicroelectronics.
 * All rights reserved.</center></h2>
 *
 * This software component is licensed by ST under BSD 3-Clause license,
 * the «License»; You may not use this file except in compliance with the
 * License. You may obtain a copy of the License at:
 *
 * opensource.org/licenses/BSD-3-Clause
 */
```

```

*

*/
/* USER CODE END Header */

/* Includes -----*/
#include «main.h»
#define led0 GPIO_PIN_0
#define led1 GPIO_PIN_1
#define led2 GPIO_PIN_2
#define led3 GPIO_PIN_3
#define led4 GPIO_PIN_4
#define led5 GPIO_PIN_5
#define led6 GPIO_PIN_6
#define led7 GPIO_PIN_7
int LEDS[] = {led0, led1, led2, led3, led4, led5, led6, led7};

void SystemClock_Config(void);
static void MX_GPIO_Init(void);

int main(void)
{
 int j;

 HAL_Init();

 /* Configure the system clock */
 SystemClock_Config();

 /* Initialize all configured peripherals */
 MX_GPIO_Init();

 /* USER CODE BEGIN WHILE */
 while (1)
 {
 for(j = 0; j <= 7; j++)
 {
 HAL_GPIO_WritePin(GPIOC, LEDS[j], GPIO_PIN_SET);
 HAL_Delay(250);
 HAL_GPIO_WritePin(GPIOC, LEDS[j], GPIO_PIN_RESET);
 HAL_Delay(250);
 }
 }
}

void SystemClock_Config(void)

```

```
{
 RCC_OscInitTypeDef RCC_OscInitStruct = {0};
 RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

 RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
 RCC_OscInitStruct.HSIState = RCC_HSI_ON;
 RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
 RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
 RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
 RCC_OscInitStruct.PLL.PLLM = 2;
 RCC_OscInitStruct.PLL.PLLN = 20;
 RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV7;
 RCC_OscInitStruct.PLL.PLLQ = RCC_PLLQ_DIV2;
 RCC_OscInitStruct.PLL.PLLR = RCC_PLLR_DIV2;
 if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
 {
 Error_Handler();
 }

 RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
 |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
 RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
 RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
 RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
 RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

 if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_4) != HAL_OK)
 {
 Error_Handler();
 }

 if (HAL_PWREx_ControlVoltageScaling(PWR_REGULATOR_VOLTAGE_SCALE1) != HAL_OK)
 {
 Error_Handler();
 }
}

static void MX_GPIO_Init(void)
{
 GPIO_InitTypeDef GPIO_InitStruct = {0};

 /* GPIO Ports Clock Enable */
 __HAL_RCC_GPIOC_CLK_ENABLE();

 /*Configure GPIO pin Output Level */
 HAL_GPIO_WritePin(GPIOC, GPIO_PIN_0|GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3
```

```

 |GPIO_PIN_4|GPIO_PIN_5|GPIO_PIN_6|GPIO_PIN_7, GPIO_PIN_RE-
SET);

/*Configure GPIO pins : PC0 PC1 PC2 PC3
 PC4 PC5 PC6 PC7 */
GPIO_InitStruct.Pin = GPIO_PIN_0|GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3
 |GPIO_PIN_4|GPIO_PIN_5|GPIO_PIN_6|GPIO_PIN_7;
GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
GPIO_InitStruct.Pull = GPIO_NOPULL;
GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);

}

void Error_Handler(void)
{
}

#ifdef USE_FULL_ASSERT

void assert_failed(uint8_t *file, uint32_t line)
{
}

#endif

/***** (C) COPYRIGHT STMicroelectronics *****/

```

*Figure 5.30: Program: MovingLEDs.*

## 5.5 Project 4: Binary Up Counter with LEDs

### Description

In this project 8 LEDs are connected to GPIO ports PC00 to PC7 as in Project 3. The program counts up from 0 to 255 continuously and turns ON/OFF the appropriate LEDs to show the count. A 1-second delay is inserted between each count.

### The aim

The aim of this project is to show how a number of port pins can be grouped together and then accessed as a group. Here, 8 pins are grouped together and accessed as a single byte. PC0 is assumed to be the LSB bit and PC7 is assumed to be the MSB bit.

### Block diagram

The block diagram of the project is as in Figure 5.27.

### Circuit diagram

The circuit diagram of the project is given in Figure 5.28.



## The construction

The project is built on a breadboard as shown in Figure 5.29.

## Program listing

Create a program as in the previous project and name the program **CountUp**. Configure port pins PC0 to PC7 as digital outputs as in Project 3 and also configure the MCU clock to operate at 80 MHz.

Figure 5.31 shows the program listing. Here, the **math.h** library is included in the project since we will be using the **pow()** function. At the beginning of the program the LED connections are configured using variables led0 to led7 as in Project 3.

Function called **Display** is created to group the GPIO pins together so that they can be accessed easily. This function has two arguments: **No** and **L**. **No** is the binary number to be displayed by the LEDs. **L** is the number of bits grouped together (8 in this project).

The program is executed forever in the following while loop where variable **Count** stores the current value of the count which is incremented every second until it reaches 255. After this point variable **Count** is reset to 0 and this process continues forever until the program is stopped by the user:

```

 while (1)
 {
 Display(Count, 8); // Display Count
 if(Count == 255) // If 255
 Count = 0; // Reset to 0
 else // If not 255
 Count++; // Increment Count
 HAL_Delay(1000); // Wait 1 second
 }

/* USER CODE BEGIN Header */
/**

 * @file : main.c
 * @brief : Main program body

 * @attention
 *
 * <h2><center>© Copyright (c) 2020 STMicroelectronics.
 * All rights reserved.</center></h2>
 *
 * This software component is licensed by ST under BSD 3-Clause license,
 * the «License»; You may not use this file except in compliance with the
 * License. You may obtain a copy of the License at:
 *
 * opensource.org/licenses/BSD-3-Clause
 *

 */

```

```
#include «main.h»
#include «math.h»

#define led0 GPIO_PIN_0
#define led1 GPIO_PIN_1
#define led2 GPIO_PIN_2
#define led3 GPIO_PIN_3
#define led4 GPIO_PIN_4
#define led5 GPIO_PIN_5
#define led6 GPIO_PIN_6
#define led7 GPIO_PIN_7
int LEDS[] = {led7, led6, led5, led4, led3, led2, led1, led0};

void SystemClock_Config(void);
static void MX_GPIO_Init(void);

//
// Group the port pins together. L is the number of bits (8 here)
// and No is the data to be displayed
//
void Display(int No, int L)
{
 int j, m, i;
 m = L - 1;
 for(i = 0; i < L; i++)
 {
 j = pow(2.0, m);
 if((No & j) != 0)
 HAL_GPIO_WritePin(GPIOC, LEDS[i], GPIO_PIN_SET);
 else
 HAL_GPIO_WritePin(GPIOC, LEDS[i], GPIO_PIN_RESET);
 m--;
 }
}

int main(void)
{
 int Count = 0;

 HAL_Init();

 SystemClock_Config();

 MX_GPIO_Init();
```

```
/* USER CODE BEGIN WHILE */
while (1)
{
 Display(Count, 8); // Display Count
 if(Count == 255) // If 255
 Count = 0; // Reset to 0
 else // If not 255
 Count++; // Increment Count
 HAL_Delay(1000); // Wait 1 second
}
}

void SystemClock_Config(void)
{
 RCC_OscInitTypeDef RCC_OscInitStruct = {0};
 RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

 RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
 RCC_OscInitStruct.HSISState = RCC_HSI_ON;
 RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
 RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
 RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
 RCC_OscInitStruct.PLL.PLLM = 2;
 RCC_OscInitStruct.PLL.PLLN = 20;
 RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV7;
 RCC_OscInitStruct.PLL.PLLQ = RCC_PLLQ_DIV2;
 RCC_OscInitStruct.PLL.PLLR = RCC_PLLR_DIV2;
 if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
 {
 Error_Handler();
 }

 RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
 |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
 RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
 RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
 RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
 RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

 if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_4) != HAL_OK)
 {
 Error_Handler();
 }
}
```

```

 if (HAL_PWREx_ControlVoltageScaling(PWR_REGULATOR_VOLTAGE_SCALE1) != HAL_OK)
 {
 Error_Handler();
 }
}

static void MX_GPIO_Init(void)
{
 GPIO_InitTypeDef GPIO_InitStruct = {0};

 /* GPIO Ports Clock Enable */
 __HAL_RCC_GPIOC_CLK_ENABLE();

 /*Configure GPIO pin Output Level */
 HAL_GPIO_WritePin(GPIOC, GPIO_PIN_0|GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3
 |GPIO_PIN_4|GPIO_PIN_5|GPIO_PIN_6|GPIO_PIN_7, GPIO_PIN_RE-
SET);

 /*Configure GPIO pins : PC0 PC1 PC2 PC3
 PC4 PC5 PC6 PC7 */
 GPIO_InitStruct.Pin = GPIO_PIN_0|GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3
 |GPIO_PIN_4|GPIO_PIN_5|GPIO_PIN_6|GPIO_PIN_7;
 GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
 GPIO_InitStruct.Pull = GPIO_NOPULL;
 GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
 HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);
}

void Error_Handler(void)
{
}

#ifdef USE_FULL_ASSERT

void assert_failed(uint8_t *file, uint32_t line)
{
}

#endif

/***** (C) COPYRIGHT STMicroelectronics *****END OF FILE*****/

```

*Figure 5.31: Program: CountUp.*

Figure 5.32 shows the LED pattern when the program is run.

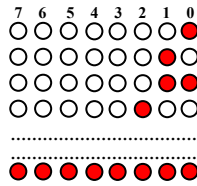


Figure 5.32: The LED pattern.

## 5.6 Project 5: Random Flashing LEDs

### Description

In this project, 8 LEDs are connected to GPIO ports PC0 to PC7 of the Nucleo-L476RG board as in Project 3. The program flashes the LEDs randomly so that they look like flashing Christmas lights.

### The aim

The aim of this project is to show how random numbers can be generated.

### Block diagram

The block diagram of the project is as in Figure 5.27.

### Circuit diagram

The circuit diagram of the project is as in Figure 5.28.

### The construction

The project is built on a breadboard as shown in Figure 5.29.

### Program listing

Create a program as in the previous project and name the program **RandomFlash**. Configure port pins PC0 to PC7 as digital outputs as in Project 3, as well as configure the MCU clock to operate at 80 MHz.

Figure 5.33 shows the program listing (the comments are not shown for clarity). Here, the **math.h** library is included in the project since we will be using the **pow()** function. Also, the **stdlib.h** library is included since we will be using the random function generator **rand()**. At the beginning of the program the LED connections are configured using variables **led0** to **led7** as in Project 3. Function called **Display** is created to group the GPIO pins together as in Project no. 3.

The program is executed forever in the following while loop where built-in function **rand()** is called to generate a random number. Thus number is then limited to be between 0 and 255. Function **Display** is called to display the number by the LEDs as in Project 3:

```
while (1)
{
 r = rand() % 255; // Random number 0 - 255
 Display(r, 8); // Display
```

```

 HAL_Delay(250); // 250ms delay
 }

/* USER CODE BEGIN Header */
/**

 * @file : main.c
 * @brief : Main program body

 * @attention
 *
 * <h2><center>© Copyright (c) 2020 STMicroelectronics.
 * All rights reserved.</center></h2>
 *
 * This software component is licensed by ST under BSD 3-Clause license,
 * the «License»; You may not use this file except in compliance with the
 * License. You may obtain a copy of the License at:
 *
 * opensource.org/licenses/BSD-3-Clause
 *

 */
#include «main.h»
#include «math.h»
#include «stdlib.h»

#define led0 GPIO_PIN_0
#define led1 GPIO_PIN_1
#define led2 GPIO_PIN_2
#define led3 GPIO_PIN_3
#define led4 GPIO_PIN_4
#define led5 GPIO_PIN_5
#define led6 GPIO_PIN_6
#define led7 GPIO_PIN_7
int LEDS[] = {led0, led1, led2, led3, led4, led5, led6, led7};

void SystemClock_Config(void);
static void MX_GPIO_Init(void);

//
// Group the port pins together. L is the number of bits (8 here)
// and No is the data to be displayed
//
void Display(int No, int L)
{
 int j, m, i;
 m = L - 1;

```

```
 for(i = 0; i < L; i++)
 {
 j = pow(2.0, m);
 if((No & j) != 0)
 HAL_GPIO_WritePin(GPIOC, LEDS[i], GPIO_PIN_SET);
 else
 HAL_GPIO_WritePin(GPIOC, LEDS[i], GPIO_PIN_RESET);
 m--;
 }
 }

/* Start of main program */
int main(void)
{
 int r;

 HAL_Init();

 SystemClock_Config();

 MX_GPIO_Init();

 while (1)
 {
 r = rand() % 255; // Random number 0 - 255
 Display(r, 8); // Display
 HAL_Delay(250); // 250ms delay
 }
}

void SystemClock_Config(void)
{
 RCC_OscInitTypeDef RCC_OscInitStruct = {0};
 RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

 RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
 RCC_OscInitStruct.HSISState = RCC_HSI_ON;
 RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
 RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
 RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
 RCC_OscInitStruct.PLL.PLLM = 2;
 RCC_OscInitStruct.PLL.PLLN = 20;
 RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV7;
 RCC_OscInitStruct.PLL.PLLQ = RCC_PLLQ_DIV2;
 RCC_OscInitStruct.PLL.PLLR = RCC_PLLR_DIV2;
```

```

if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
{
 Error_Handler();
}

RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
 |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_4) != HAL_OK)
{
 Error_Handler();
}

if (HAL_PWREx_ControlVoltageScaling(PWR_REGULATOR_VOLTAGE_SCALE1) != HAL_OK)
{
 Error_Handler();
}
}

static void MX_GPIO_Init(void)
{
 GPIO_InitTypeDef GPIO_InitStruct = {0};

 /* GPIO Ports Clock Enable */
 __HAL_RCC_GPIOC_CLK_ENABLE();

 /*Configure GPIO pin Output Level */
 HAL_GPIO_WritePin(GPIOC, GPIO_PIN_0|GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3
 |GPIO_PIN_4|GPIO_PIN_5|GPIO_PIN_6|GPIO_PIN_7, GPIO_PIN_RE-
SET);

 /*Configure GPIO pins : PC0 PC1 PC2 PC3
 PC4 PC5 PC6 PC7 */
 GPIO_InitStruct.Pin = GPIO_PIN_0|GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3
 |GPIO_PIN_4|GPIO_PIN_5|GPIO_PIN_6|GPIO_PIN_7;
 GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
 GPIO_InitStruct.Pull = GPIO_NOPULL;
 GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
 HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);
}

```



```
void Error_Handler(void)
{
}

#ifdef USE_FULL_ASSERT

void assert_failed(uint8_t *file, uint32_t line)
{
}
#endif

/***** (C) COPYRIGHT STMicroelectronics *****/
```

*Figure 5.33 Program: RandomFlash.*

## 5.7 Project 6: Pushbutton and LED

### Description

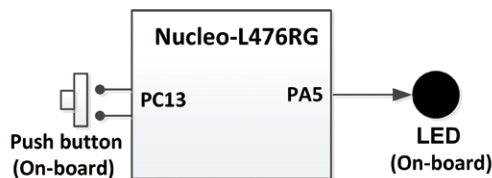
In this project the user pushbutton and the user LED on the Nucleo-L476RG board are used. The LED is turned ON whenever the button is pressed.

### The aim

The aim of this project is to show how the state of a button can be read in a program.

### Block diagram

The block diagram of the project is shown in Figure 5.34 where the on-board button and the LED are used.



*Figure 5.34: Block diagram of the project.*

### Circuit diagram

The circuit diagram of the project is shown in Figure 5.35. User button is connected to GPIO pin PC13 and is normally pulled high through a 4.7-kohm resistor. The state of the button is normally logic 1, and when the button is pressed the state of the button goes from logic 1 to logic 0.

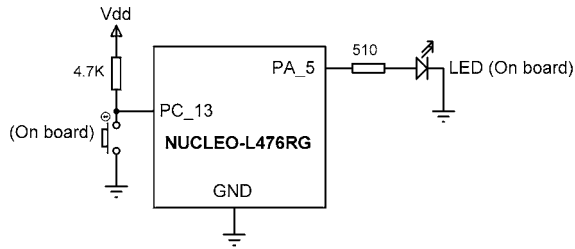


Figure 5.35: Circuit diagram of the project.

### Program listing

The steps to create the program are as follows:

- start the STM32CubeIDE;
- create a new workspace or accept the default one;
- enter the MCU type as STM32L476RG;
- give a name to the project, e.g. **BUTTON**;
- click pin **PA5** and configure it as digital output as before;
- click pin **PC13** and configure it as digital input (**GPIO\_Input**) as shown in Figure 5.36;

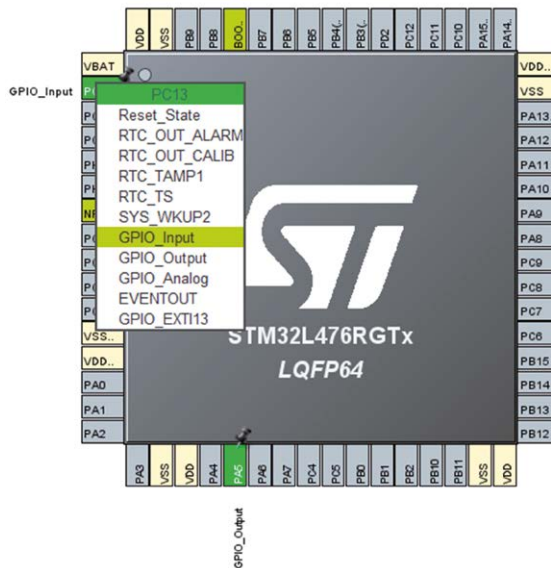


Figure 5.36: Configure PC13 as digital input.

- configure the MCU clock for 80MHz as in the previous projects;
- click **File** and then **Save** and then click **YES** to generate code.

Figure 5.37 shows the program listing (the comments are not shown for clarity). At the beginning of the program **myled** and **mybutton** are assigned to **GPIO\_PIN\_5** (on-board user LED) and to **GPIO\_PIN\_13** (on-board user button) as shown below:

```
#define myled GPIO_PIN_5
#define mybutton GPIO_PIN_13
```

Inside the **MX\_GPIO\_Init** function clocks are enabled for GPIOA and GPIOC, **GPIO\_PIN\_5** is configured as output and **GPIO\_PIN\_13** is configured as input as shown below:

```
static void MX_GPIO_Init(void)
{
 GPIO_InitTypeDef GPIO_InitStruct = {0};

 /* GPIO Ports Clock Enable */
 __HAL_RCC_GPIOC_CLK_ENABLE();
 __HAL_RCC_GPIOA_CLK_ENABLE();

 /*Configure GPIO pin Output Level */
 HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_RESET);

 /*Configure GPIO pin : PC13 */
 GPIO_InitStruct.Pin = GPIO_PIN_13;
 GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
 GPIO_InitStruct.Pull = GPIO_NOPULL;
 HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);

 /*Configure GPIO pin : PA5 */
 GPIO_InitStruct.Pin = GPIO_PIN_5;
 GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
 GPIO_InitStruct.Pull = GPIO_NOPULL;
 GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
 HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
}
```

The program code to check the state of the button and to control the LED is shown below. Initially the LED is turned OFF outside the program loop. Inside the while loop the state of the button is checked and if the button is pressed (button state = 0) then the LED is turned ON, otherwise the LED is turned OFF:

```
HAL_GPIO_WritePin(GPIOA, myled, GPIO_PIN_RESET);
while (1)
{
 if(HAL_GPIO_ReadPin(GPIOC, mybutton) == 0)
 HAL_GPIO_WritePin(GPIOA, myled, GPIO_PIN_SET);
 else
 HAL_GPIO_WritePin(GPIOA, myled, GPIO_PIN_RESET);
}
```

Compile the program and make sure there are no errors. Upload the binary file **BUTTON.bin** by dragging and dropping it to device NUCLEO-L476RG.

```

/* USER CODE BEGIN Header */
/**

 * @file : main.c
 * @brief : Main program body

 * @attention
 *
 * <h2><center>© Copyright (c) 2020 STMicroelectronics.
 * All rights reserved.</center></h2>
 *
 * This software component is licensed by ST under BSD 3-Clause license,
 * the «License»; You may not use this file except in compliance with the
 * License. You may obtain a copy of the License at:
 *
 * opensource.org/licenses/BSD-3-Clause
 *

 */
#include «main.h»

#define myled GPIO_PIN_5
#define mybutton GPIO_PIN_13

void SystemClock_Config(void);
static void MX_GPIO_Init(void);

int main(void)
{
 HAL_Init();

 SystemClock_Config();

 MX_GPIO_Init();

 HAL_GPIO_WritePin(GPIOA, myled, GPIO_PIN_RESET); // LED OFF
 while (1)
 {
 if(HAL_GPIO_ReadPin(GPIOC, mybutton) == 0) // IF Button pressed
 HAL_GPIO_WritePin(GPIOA, myled, GPIO_PIN_SET); // LED ON
 else
 HAL_GPIO_WritePin(GPIOA, myled, GPIO_PIN_RESET); // LED OFF
 }
}

```

```
}

void SystemClock_Config(void)
{
 RCC_OscInitTypeDef RCC_OscInitStruct = {0};
 RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

 RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
 RCC_OscInitStruct.HSISState = RCC_HSI_ON;
 RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
 RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
 RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
 RCC_OscInitStruct.PLL.PLLM = 2;
 RCC_OscInitStruct.PLL.PLLN = 20;
 RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV7;
 RCC_OscInitStruct.PLL.PLLQ = RCC_PLLQ_DIV2;
 RCC_OscInitStruct.PLL.PLLR = RCC_PLLR_DIV2;
 if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
 {
 Error_Handler();
 }

 RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
 |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
 RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
 RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
 RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
 RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

 if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_4) != HAL_OK)
 {
 Error_Handler();
 }

 if (HAL_PWREx_ControlVoltageScaling(PWR_REGULATOR_VOLTAGE_SCALE1) != HAL_OK)
 {
 Error_Handler();
 }
}

static void MX_GPIO_Init(void)
{
 GPIO_InitTypeDef GPIO_InitStruct = {0};

 /* GPIO Ports Clock Enable */
```

```

__HAL_RCC_GPIOC_CLK_ENABLE();
__HAL_RCC_GPIOA_CLK_ENABLE();

/*Configure GPIO pin Output Level */
HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_RESET);

/*Configure GPIO pin : PC13 */
GPIO_InitStruct.Pin = GPIO_PIN_13;
GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
GPIO_InitStruct.Pull = GPIO_NOPULL;
HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);

/*Configure GPIO pin : PA5 */
GPIO_InitStruct.Pin = GPIO_PIN_5;
GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
GPIO_InitStruct.Pull = GPIO_NOPULL;
GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
}

void Error_Handler(void)
{
}

#ifdef USE_FULL_ASSERT

void assert_failed(uint8_t *file, uint32_t line)
{
}
#endif

/***** (C) COPYRIGHT STMicroelectronics *****/

```

*Figure 5.37: Program: BUTTON.*

## 5.8 Project 7: Control of Multiple LEDs by 2 Buttons

### Description

In this project 8 LEDs are connected to GPIO ports PC0 to PC7 of the Nucleo-L476RG board as in Project 3. Additionally, 2 buttons named **LEFT** and **RIGHT** are connected to port pins PA13 and PA14 respectively. Initially, the LED at the LSB position is lit. Each time the **LEFT** button is pressed, the LED towards the MSB is lit. Similarly, each time the **RIGHT** button is pressed, the LED towards the LSB is lit.

### The aim

The aim of this project is to show how external buttons can be used in projects.

## Block diagram

The block diagram of the project is as in Figure 5.38 with 2 buttons as inputs and 8 LEDs as outputs.

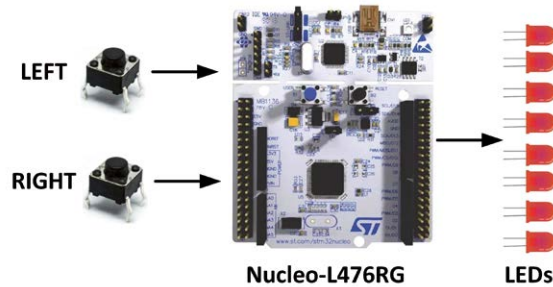


Figure 5.38: Block diagram of the project.

## Circuit diagram

The circuit diagram of the project appears in Figure 5.39. The LEDs are connected as in Project 3. One terminal of the buttons is connected to GND and the other terminal to PA13 and PA14. These pins are configured as pull-up in software so that their normal state is at logic 1 and they go to logic 0 when the button is pressed.

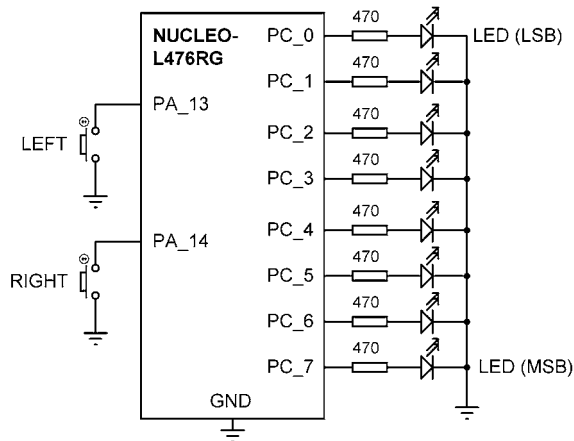
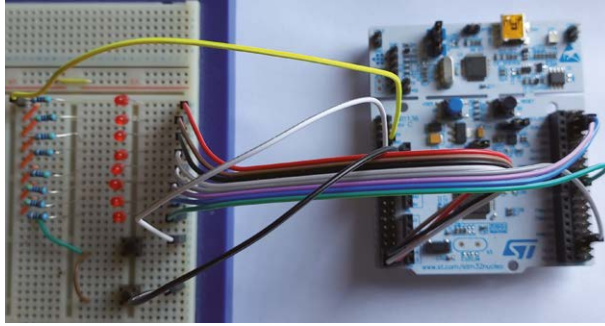


Figure 5.39: Circuit diagram of the project.

## The construction

The project is built on a breadboard as shown in Figure 5.40. Connections between the development board and the breadboard are made using jumper wires.



*Figure 5.40: Project constructed on a breadboard.*

## Program listing

The steps to create the program are as follows:

- start the STM32CubeIDE;
- create a new workspace or accept the default one;
- enter the MCU type as STM32L476RG;
- give a name to the project, e.g. **BUTTONS**;
- configure **PC0** to **PC7** as digital outputs (**GPIO\_Output**);
- configure **PA13** and **PA14** as digital inputs (**GPIO\_Input**) as in Figure 5.41;

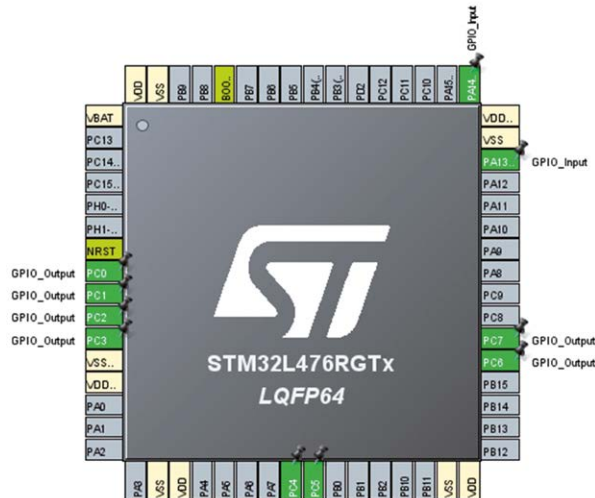


Figure 5.41: Configure inputs and outputs.

- we now have to enable the pull-up resistors at the two inputs;
- click **System Core**, followed by **GPIO**;
- click **PA13a** and set the **GPIO Pull-up/Pull-down** to **Pull-up** as shown in Figure 5.42. Do the same for PA14;



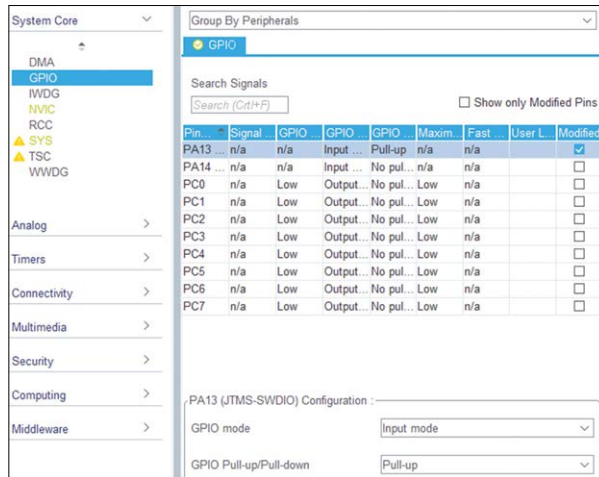


Figure 5.42: Enable the pull-up resistors at inputs.

- click tab **Clock Configuration** and set the MCU clock to 80 MHz;
- click **File** and then **Save** and **YES** to generate code;
- click **Core**, followed by **Src** and then double click **main.c** to open the main program.

Figure 5.42 shows the program listing (the comments are not shown for clarity). At the beginning of the program the LED and button connections are defined as follows:

```
#define led0 GPIO_PIN_0
#define led1 GPIO_PIN_1
#define led2 GPIO_PIN_2
#define led3 GPIO_PIN_3
#define led4 GPIO_PIN_4
#define led5 GPIO_PIN_5
#define led6 GPIO_PIN_6
#define led7 GPIO_PIN_7
int LEDS[] = {led0, led1, led2, led3, led4, led5, led6, led7};

#define LEFT GPIO_PIN_13
#define RIGHT GPIO_PIN_14
```

The main program loop starts with the while statement. Just before entering this loop LED at the LSB position is turned ON. Inside the loop the state of button LEFT is checked. If the button is pressed, then the current LED is turned OFF and the LED on its left (towards the MSB) is turned ON. The same test is done for button RIGHT. If the button is pressed, then the current LED is turned OFF and the LED on its right (towards the LSB) is turned ON.

A small delay is inserted and the program checks if a pressed button is released or not. This is done in order to eliminate any contact bouncing problems associated with mechanical relays. The code that is executed inside the loop is:

```

while (1)
{
 if(HAL_GPIO_ReadPin(GPIOA, LEFT) == 0)
 {
 HAL_GPIO_WritePin(GPIOC, LEDS[j], GPIO_PIN_RESET);
 j++;
 if(j == 8)j = 0;
 HAL_GPIO_WritePin(GPIOC, LEDS[j], GPIO_PIN_SET);
 HAL_Delay(100);
 while(HAL_GPIO_ReadPin(GPIOA, LEFT) == 0);
 }

 if(HAL_GPIO_ReadPin(GPIOA, RIGHT) == 0)
 {
 HAL_GPIO_WritePin(GPIOC, LEDS[j], GPIO_PIN_RESET);
 j--;
 if(j < 0)j = 7;
 HAL_GPIO_WritePin(GPIOC, LEDS[j], GPIO_PIN_SET);
 HAL_Delay(100);
 while(HAL_GPIO_ReadPin(GPIOA, RIGHT) == 0);
 }
}

```

Compile the program and make sure there are no errors. Upload the binary file **BUTTONS.bin** by dragging and dropping it to device NUCLEO-L476RG.

```

/* USER CODE BEGIN Header */
/**
 * *****
 * @file : main.c
 * @brief : Main program body
 * *****
 * @attention
 *
 * <h2><center>© Copyright (c) 2020 STMicroelectronics.
 * All rights reserved.</center></h2>
 *
 * This software component is licensed by ST under BSD 3-Clause license,
 * the «License»; You may not use this file except in compliance with the
 * License. You may obtain a copy of the License at:
 *
 * opensource.org/licenses/BSD-3-Clause
 *
 * *****
 */
#include «main.h»

```

```
#define led0 GPIO_PIN_0
#define led1 GPIO_PIN_1
#define led2 GPIO_PIN_2
#define led3 GPIO_PIN_3
#define led4 GPIO_PIN_4
#define led5 GPIO_PIN_5
#define led6 GPIO_PIN_6
#define led7 GPIO_PIN_7
int LEDS[] = {led0, led1, led2, led3, led4, led5, led6, led7};

#define LEFT GPIO_PIN_13
#define RIGHT GPIO_PIN_14

void SystemClock_Config(void);
static void MX_GPIO_Init(void);

int main(void)
{
 int j = 0;
 HAL_Init();

 SystemClock_Config();

 MX_GPIO_Init();

 HAL_GPIO_WritePin(GPIOC, LEDS[j], GPIO_PIN_SET); // LED LSB ON
 while (1)
 {
 if(HAL_GPIO_ReadPin(GPIOA, LEFT) == 0) // If LEFT pressed
 {
 HAL_GPIO_WritePin(GPIOC, LEDS[j], GPIO_PIN_RESET); // Current LED
OFF
 j++;
 if(j == 8)j = 0; // If at MSB position
 HAL_GPIO_WritePin(GPIOC, LEDS[j], GPIO_PIN_SET); // Next LED ON
 HAL_Delay(100); // Contact debouncing
 while(HAL_GPIO_ReadPin(GPIOA, LEFT) == 0); // Contact debouncing
 }

 if(HAL_GPIO_ReadPin(GPIOA, RIGHT) == 0) // If RIGHT pressed
 {
 HAL_GPIO_WritePin(GPIOC, LEDS[j], GPIO_PIN_RESET); // Current LED
OFF
 j--;
 if(j < 0)j = 7; // If at LSB position
 HAL_GPIO_WritePin(GPIOC, LEDS[j], GPIO_PIN_SET); // Previous LED ON
```

```

 HAL_Delay(100); // Contact debouncing
 while(HAL_GPIO_ReadPin(GPIOA, RIGHT) == 0); // Contact debouncing
 }
}

void SystemClock_Config(void)
{
 RCC_OscInitTypeDef RCC_OscInitStruct = {0};
 RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

 RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
 RCC_OscInitStruct.HSISState = RCC_HSI_ON;
 RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
 RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
 RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
 RCC_OscInitStruct.PLL.PLLM = 2;
 RCC_OscInitStruct.PLL.PLLN = 20;
 RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV7;
 RCC_OscInitStruct.PLL.PLLQ = RCC_PLLQ_DIV2;
 RCC_OscInitStruct.PLL.PLLR = RCC_PLLR_DIV2;
 if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
 {
 Error_Handler();
 }

 RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
 |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
 RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
 RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
 RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
 RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

 if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_4) != HAL_OK)
 {
 Error_Handler();
 }

 if (HAL_PWREx_ControlVoltageScaling(PWR_REGULATOR_VOLTAGE_SCALE1) != HAL_OK)
 {
 Error_Handler();
 }
}

static void MX_GPIO_Init(void)
{

```

```
GPIO_InitTypeDef GPIO_InitStruct = {0};

/* GPIO Ports Clock Enable */
__HAL_RCC_GPIOC_CLK_ENABLE();
__HAL_RCC_GPIOA_CLK_ENABLE();

/*Configure GPIO pin Output Level */
HAL_GPIO_WritePin(GPIOC, GPIO_PIN_0|GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3
 |GPIO_PIN_4|GPIO_PIN_5|GPIO_PIN_6|GPIO_PIN_7, GPIO_PIN_RE-
SET);

/*Configure GPIO pins : PC0 PC1 PC2 PC3
 PC4 PC5 PC6 PC7 */
GPIO_InitStruct.Pin = GPIO_PIN_0|GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3
 |GPIO_PIN_4|GPIO_PIN_5|GPIO_PIN_6|GPIO_PIN_7;
GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
GPIO_InitStruct.Pull = GPIO_NOPULL;
GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);

/*Configure GPIO pins : PA2 PA3 */
GPIO_InitStruct.Pin = GPIO_PIN_13|GPIO_PIN_14;
GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
GPIO_InitStruct.Pull = GPIO_PULLUP;
HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);

}

void Error_Handler(void)
{
}

#ifdef USE_FULL_ASSERT

void assert_failed(uint8_t *file, uint32_t line)
{
}

#endif

/***** (C) COPYRIGHT STMicroelectronics *****/
```

*Figure 5.42: Program: BUTTONS.*

## 5.9 Project 8: LED Dice

### Description

In this project an electronic dice is designed with LEDs and a pushbutton switch. The LEDs are organized to simulate the look of the faces of a real physical dice. When the button is pressed a random number is generated between 1 and 6 and is displayed on the LEDs.

### The aim

The aim of this project is to show how an LED-based dice can be designed using the Nucleo-L476 board.

### Block diagram

The block diagram of the project is shown in Figure 5.43 where the on-board button and external 7 LEDs are used in the project.

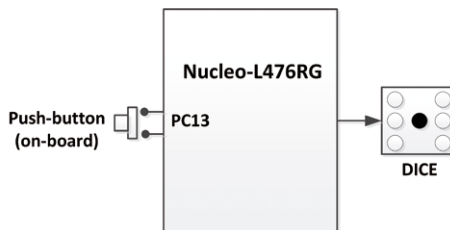


Figure 5.43: Block diagram of the project.

As shown in Figure 5.44, the LEDs are organized such that they can indicate the numbers as in a real dice. Operation of the project is as follows: Normally all the LEDs are OFF to indicate that the system is ready. Pressing the button generates a new dice number which is displayed on the LEDs for 3 seconds. After this time all the LEDs turn OFF again.

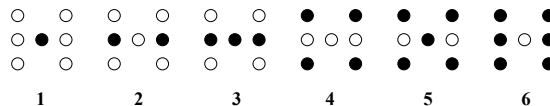


Figure 5.44: LED dice.

### Circuit diagram

The circuit diagram of the project is shown in Figure 5.45. The 7 LEDs are connected to GPIO pins through 470-ohm resistors as follows:

| LED | GPIO pin |
|-----|----------|
| D1  | PC0      |
| D2  | PC1      |
| D3  | PC2      |
| D4  | PC3      |
| D5  | PC4      |
| D6  | PC5      |
| D7  | PC6      |

The on-board pushbutton switch (at port pin PC13) is used in the project to start a game.

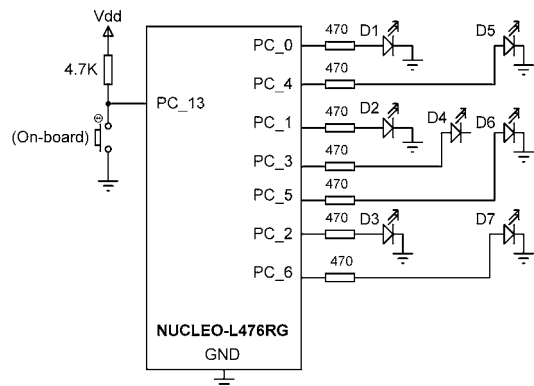


Figure 5.45: Circuit diagram of the project.

**PDL of the project**

Figure 5.46 shows the operation of the project as a PDL. The relationship between the number to be displayed and the LEDs to be turned ON is shown in Table 5.1. For example, to display number 1, only LED D4 must be turned ON. Similarly, to display number 3, LEDs D2, D4 and D6 must be turned ON, and so on.

| Required Number | LEDs to be Turned ON   |
|-----------------|------------------------|
| 1               | D4                     |
| 2               | D2, D6                 |
| 3               | D2, D4, D6             |
| 4               | D1, D3, D5, D7         |
| 5               | D1, D3, D4, D5, D7     |
| 6               | D1, D2, D3, D5, D6, D7 |

Table 5.1 Dice numbers and LEDs to be turned ON

```
BEGIN
 Assign Button to PC13
 Assign LEDs to the GPIOs
 Configure all LEDs as outputs
 Configure Button as input
 DO FOREVER
 Turn OFF all LEDs
 Wait until the Button is pressed
 Generate a random number between 1 and 6
 Turn ON the appropriate LED to display the number
 Wait for 3 seconds
 ENDDO
END
```

Figure 5.46 PDL of the project

## Program listing

The program is named **DICE**. Configure PC0 to PC6 as digital outputs and PC\_13 as digital input (see Figure 5.47).

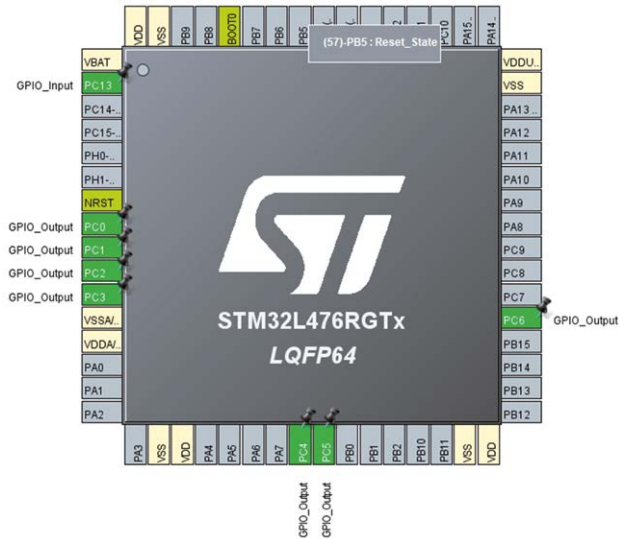


Figure 5.47: Configure the GPIO.

Figure 5.48 shows the program listing, where the comments are not shown for clarity. At the beginning of the program **stdlib.h** and **time.h** header files are added to the program ('srand' and 'rand' functions require **stdlib.h**. Array LEDs are then created to store the GPIO pin names of the pins where the LEDs are connected to. Also, the button stores the pin number 13:

```
#define led1 GPIO_PIN_0
#define led2 GPIO_PIN_1
#define led3 GPIO_PIN_2
#define led4 GPIO_PIN_3
#define led5 GPIO_PIN_4
#define led6 GPIO_PIN_5
#define led7 GPIO_PIN_6
int LEDs[] = {led1, led2, led3, led4, led5, led6, led7};
#define button GPIO_PIN_13
```

Function **MX\_GPIO\_Init** is modified to configure the LED ports as outputs and the button port as input:

```
static void MX_GPIO_Init(void)
{
 GPIO_InitTypeDef GPIO_InitStruct = {0};
```



```

/* GPIO Ports Clock Enable */
__HAL_RCC_GPIOC_CLK_ENABLE();

/*Configure GPIO pin Output Level */
HAL_GPIO_WritePin(GPIOC, GPIO_PIN_0|GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3
 |GPIO_PIN_4|GPIO_PIN_5|GPIO_PIN_6, GPIO_PIN_RE-
SET);

/*Configure GPIO pin : PC13 */
GPIO_InitStruct.Pin = GPIO_PIN_13;
GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
GPIO_InitStruct.Pull = GPIO_NOPULL;
HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);

/*Configure GPIO pins : PC0 PC1 PC2 PC3
 PC4 PC5 PC6 */
GPIO_InitStruct.Pin = GPIO_PIN_0|GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3
 |GPIO_PIN_4|GPIO_PIN_5|GPIO_PIN_6;
GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
GPIO_InitStruct.Pull = GPIO_NOPULL;
GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);
}

```

At the beginning of the main program function **srand** is called to set the seed value for the random number generator. Inside the main program loop all the LEDs are turned OFF by calling to function **ALL\_OFF** and then the program waits until the button is pressed. After the button is pressed, a new random number is generated between 1 and 6 and then a switch statement is used to turn ON the appropriate LEDs so that the generated dice number can be shown on the LEDs. The generated number is displayed for 3 seconds. After this time, all the LEDs are turned OFF and the process is repeated.

The code to generate a random number and control the LEDs is as follows:

```

srand(time(NULL));
while (1)
{
 ALL_OFF();
 while(HAL_GPIO_ReadPin(GPIOC, button) == 1);
 dice = rand()%6 + 1;
 switch(dice)
 {
 case 1:
 HAL_GPIO_WritePin(GPIOA,LEDS[3], GPIO_PIN_SET);
 break;
 case 2:

```

```

 HAL_GPIO_WritePin(GPIOA,LEDS[1], GPIO_PIN_SET);
 HAL_GPIO_WritePin(GPIOA,LEDS[5], GPIO_PIN_SET);
 break;
 case 3:
 HAL_GPIO_WritePin(GPIOA,LEDS[1], GPIO_PIN_SET);
 HAL_GPIO_WritePin(GPIOA,LEDS[3], GPIO_PIN_SET);
 HAL_GPIO_WritePin(GPIOA,LEDS[5], GPIO_PIN_SET);
 break;
 case 4:
 HAL_GPIO_WritePin(GPIOA,LEDS[0], GPIO_PIN_SET);
 HAL_GPIO_WritePin(GPIOA,LEDS[2], GPIO_PIN_SET);
 HAL_GPIO_WritePin(GPIOA,LEDS[4], GPIO_PIN_SET);
 HAL_GPIO_WritePin(GPIOA,LEDS[6], GPIO_PIN_SET);
 break;
 case 5:
 HAL_GPIO_WritePin(GPIOA,LEDS[0], GPIO_PIN_SET);
 HAL_GPIO_WritePin(GPIOA,LEDS[2], GPIO_PIN_SET);
 HAL_GPIO_WritePin(GPIOA,LEDS[3], GPIO_PIN_SET);
 HAL_GPIO_WritePin(GPIOA,LEDS[4], GPIO_PIN_SET);
 HAL_GPIO_WritePin(GPIOA,LEDS[6], GPIO_PIN_SET);
 break;
 case 6:
 HAL_GPIO_WritePin(GPIOA,LEDS[0], GPIO_PIN_SET);
 HAL_GPIO_WritePin(GPIOA,LEDS[1], GPIO_PIN_SET);
 HAL_GPIO_WritePin(GPIOA,LEDS[2], GPIO_PIN_SET);
 HAL_GPIO_WritePin(GPIOA,LEDS[4], GPIO_PIN_SET);
 HAL_GPIO_WritePin(GPIOA,LEDS[5], GPIO_PIN_SET);
 HAL_GPIO_WritePin(GPIOA,LEDS[6], GPIO_PIN_SET);
 break;
 }
 HAL_Delay(3000);
}

```

Compile the program and make sure there are no errors. Upload the binary file **DICE.bin** by dragging and dropping it to device NUCLEO-L476RG.

```

/* USER CODE BEGIN Header */
/**
 * *****
 * @file : main.c
 * @brief : Main program body
 * *****
 * @attention
 *
 * <h2><center>© Copyright (c) 2020 STMicroelectronics.
 * All rights reserved.</center></h2>

```

```

*
* This software component is licensed by ST under BSD 3-Clause license,
* the «License»; You may not use this file except in compliance with the
* License. You may obtain a copy of the License at:
*
* opensource.org/licenses/BSD-3-Clause
*

*/
#include «main.h»
#include «stdlib.h»
#include «time.h»

#define led1 GPIO_PIN_0
#define led2 GPIO_PIN_1
#define led3 GPIO_PIN_2
#define led4 GPIO_PIN_3
#define led5 GPIO_PIN_4
#define led6 GPIO_PIN_5
#define led7 GPIO_PIN_6

int LEDS[] = {led1, led2, led3, led4, led5, led6, led7};
#define button GPIO_PIN_13

void SystemClock_Config(void);
static void MX_GPIO_Init(void);

//
// This function turns OFF all LEDs
//
void ALL_OFF()
{
 int i;
 for(i = 0; i < 7; i++)
 {
 HAL_GPIO_WritePin(GPIOC, LEDS[i], GPIO_PIN_RESET);
 }
}

int main(void)
{
 int dice;
 HAL_Init();

 SystemClock_Config();

```

```

MX_GPIO_Init();

srand(time(NULL)); // Random number seed
while (1) // DO FOREVER
{
 ALL_OFF();
 while(HAL_GPIO_ReadPin(GPIOC, button) == 1); // Wait for button
 dice = rand()%6 + 1; // Get a dice number

 switch(dice)
 {
 case 1: // IS it 1?
 HAL_GPIO_WritePin(GPIOC,LEDS[3], GPIO_PIN_SET);
 break;

 case 2: // Is it 2?
 HAL_GPIO_WritePin(GPIOC,LEDS[1], GPIO_PIN_SET);
 HAL_GPIO_WritePin(GPIOC,LEDS[5], GPIO_PIN_SET);
 break;

 case 3: // Is it 3?
 HAL_GPIO_WritePin(GPIOC,LEDS[1], GPIO_PIN_SET);
 HAL_GPIO_WritePin(GPIOC,LEDS[3], GPIO_PIN_SET);
 HAL_GPIO_WritePin(GPIOC,LEDS[5], GPIO_PIN_SET);
 break;

 case 4: // Is it 4?
 HAL_GPIO_WritePin(GPIOC,LEDS[0], GPIO_PIN_SET);
 HAL_GPIO_WritePin(GPIOC,LEDS[2], GPIO_PIN_SET);
 HAL_GPIO_WritePin(GPIOC,LEDS[4], GPIO_PIN_SET);
 HAL_GPIO_WritePin(GPIOC,LEDS[6], GPIO_PIN_SET);
 break;

 case 5: // IS it 5?
 HAL_GPIO_WritePin(GPIOC,LEDS[0], GPIO_PIN_SET);
 HAL_GPIO_WritePin(GPIOC,LEDS[2], GPIO_PIN_SET);
 HAL_GPIO_WritePin(GPIOC,LEDS[3], GPIO_PIN_SET);
 HAL_GPIO_WritePin(GPIOC,LEDS[4], GPIO_PIN_SET);
 HAL_GPIO_WritePin(GPIOC,LEDS[6], GPIO_PIN_SET);
 break;

 case 6: // Is it 6?
 HAL_GPIO_WritePin(GPIOC,LEDS[0], GPIO_PIN_SET);
 HAL_GPIO_WritePin(GPIOC,LEDS[1], GPIO_PIN_SET);
 HAL_GPIO_WritePin(GPIOC,LEDS[2], GPIO_PIN_SET);
 HAL_GPIO_WritePin(GPIOC,LEDS[4], GPIO_PIN_SET);
 HAL_GPIO_WritePin(GPIOC,LEDS[5], GPIO_PIN_SET);
 HAL_GPIO_WritePin(GPIOC,LEDS[6], GPIO_PIN_SET);
 break;
 }
 HAL_Delay(3000); // Wait 3 seconds
}

```

```
 }
}

void SystemClock_Config(void)
{
 RCC_OscInitTypeDef RCC_OscInitStruct = {0};
 RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

 RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
 RCC_OscInitStruct.HSIState = RCC_HSI_ON;
 RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
 RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
 RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
 RCC_OscInitStruct.PLL.PLLM = 2;
 RCC_OscInitStruct.PLL.PLLN = 20;
 RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV7;
 RCC_OscInitStruct.PLL.PLLQ = RCC_PLLQ_DIV2;
 RCC_OscInitStruct.PLL.PLLR = RCC_PLLR_DIV2;
 if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
 {
 Error_Handler();
 }

 RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
 |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
 RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
 RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
 RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
 RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

 if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_4) != HAL_OK)
 {
 Error_Handler();
 }

 if (HAL_PWREx_ControlVoltageScaling(PWR_REGULATOR_VOLTAGE_SCALE1) != HAL_OK)
 {
 Error_Handler();
 }
}

static void MX_GPIO_Init(void)
{
 GPIO_InitTypeDef GPIO_InitStruct = {0};

 /* GPIO Ports Clock Enable */
```

```

__HAL_RCC_GPIOC_CLK_ENABLE();

/*Configure GPIO pin Output Level */
HAL_GPIO_WritePin(GPIOC, GPIO_PIN_0|GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3
 |GPIO_PIN_4|GPIO_PIN_5|GPIO_PIN_6, GPIO_PIN_RESET);

/*Configure GPIO pin : PC13 */
GPIO_InitStruct.Pin = GPIO_PIN_13;
GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
GPIO_InitStruct.Pull = GPIO_NOPULL;
HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);

/*Configure GPIO pins : PC0 PC1 PC2 PC3
 PC4 PC5 PC6 */
GPIO_InitStruct.Pin = GPIO_PIN_0|GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3
 |GPIO_PIN_4|GPIO_PIN_5|GPIO_PIN_6;
GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
GPIO_InitStruct.Pull = GPIO_NOPULL;
GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);
}

void Error_Handler(void)
{
}

#ifdef USE_FULL_ASSERT

void assert_failed(uint8_t *file, uint32_t line)
{
}

#endif

/***** (C) COPYRIGHT STMicroelectronics *****/

```

*Figure 5.48: Program: DICE.*

### Modified program

The program given in Figure 5.48 can be modified and made more user friendly as well as more efficient if we combine the GPIO pins into a group and then send data to this group with a single instruction.

In this example we have been using bits 0 to 6 (inclusive) of GPIOC. GPIOC is a 16-bit port with pin names PC\_0 to PC\_15. Table 5.2 shows the data to be sent to GPIOC to turn ON the appropriate LEDs to display a number. In this table the unused port pins are given the values of 0. The table shows the dice numbers, GPIOC bit numbers, and the hexadecimal number that should be sent to GPIOC to display the number.

| Dice number | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Hexadecimal number |
|-------------|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|--------------------|
| 1           | 0  | 0  | 0  | 0  | 0  | 0  | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0x0008             |
| 2           | 0  | 0  | 0  | 0  | 0  | 0  | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0x0022             |
| 3           | 0  | 0  | 0  | 0  | 0  | 0  | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0x002A             |
| 4           | 0  | 0  | 0  | 0  | 0  | 0  | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0x0055             |
| 5           | 0  | 0  | 0  | 0  | 0  | 0  | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 0x005D             |
| 6           | 0  | 0  | 0  | 0  | 0  | 0  | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0x0077             |

Table 5.2: Data to be sent to GPIOC to turn ON the appropriate LEDs.

The modified program (**DICE2**) is shown in Figure 5.49. **The clock configuration and error handler routines are the same as in the previous programs and they have been removed for clarity.** The data to be sent to port GPIOC for each number is stored in hexadecimal format in array **LEDS**. Notice that since the array index starts from 0 and not 1, a dummy 0 is used as the first array element. Thus for example, if the dice number is 1 then the number sent to GPIOC is **LEDS[1]** which is 0x0008:

```
int LEDS[] = {0x0,0x0008,0x0022,0x002A,0x0055,0x005D,0x0077};
```

The main program loop gets a random number calling function **rand()** and then sends data to GPIOC after using this number as an index to array **LEDS**. The program loop is simply:

```
while (1)
{
 ALL_OFF();
 while(HAL_GPIO_ReadPin(GPIOC, button) == 1);
 dice = rand()%6 + 1;
 GPIOC->ODR = LEDS[dice];
 HAL_Delay(3000);
}
```

Figure 5.49: The DICE2 Program.

## 5.10 Project 9: 7-Segment LED Counter

### Description

This project describes the design of a single digit 7-segment LED based counter which counts from 0 to 9 continuously with one second delay between each count.

7-segment displays are used frequently in electronic circuits to show numeric or alphanumeric values. A 7-segment display basically consists of 7 LEDs (see Figure 5.50) connected such that numbers from 0 to 9 and some letters can be displayed. Segments are identified by letters from **a** to **g** and Figure 5.51 shows the segment names of a typical 7-segment LED display.



Figure 5.50: 7-segment display.

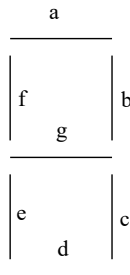


Figure 5.51: Segments of a 7-segment display.

Figure 5.52 shows how numbers from 0 to 9 can be obtained by turning ON/OFF different segments of the display.

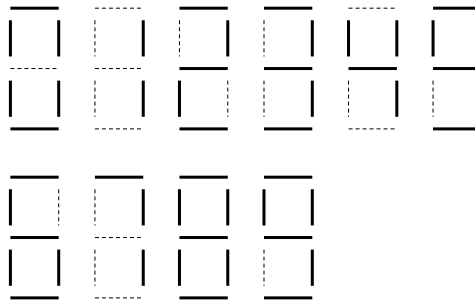


Figure 5.52: Displaying numbers 0–9.

7-segment displays are available as **common-cathode** or **common-anode**. In common-cathode configuration (see Figure 5.53) the cathodes of all the segment LEDs are connected together to ground. The individual segments are then turned ON by applying logic 1 to the required segment via current limiting resistors.



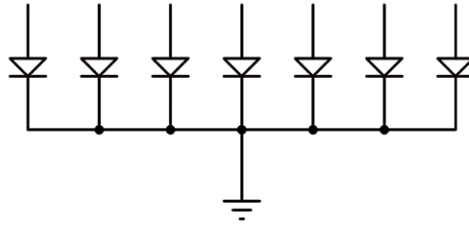


Figure 5.53: Common-cathode 7-segment display.

In a common-anode configuration, the anode terminals of all the LEDs are connected together as shown in Figure 5.54. This common point is then normally connected to the supply voltage. A segment is turned ON by connecting its cathode terminal to logic 0 via a current limiting resistor.

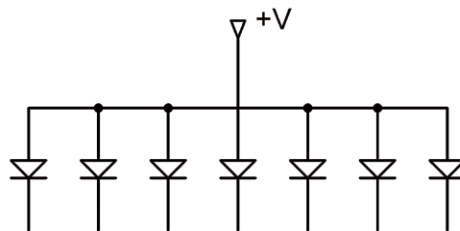


Figure 5.54: Common-anode, 7-segment display.

In this project, a type **SMA42056** model common-cathode 7-segment display is used. This is a 14.20 mm (0.56 inch) sized display that can be viewed from a distance up to 7 metres. The LED forward voltage is about 2 V. The display has 10 pins and it also has a segment LED for the decimal point. Table 5.3 shows the pin configuration of this display. The pin numbering starts from the bottom left corner of the display with pin 1 as shown in Figure 5.55. The bottom right corner is pin 5, and the top left corner is pin 10.

| Pin number | Segment        |
|------------|----------------|
| 1          | E              |
| 2          | D              |
| 3          | common cathode |
| 4          | C              |
| 5          | decimal point  |
| 6          | B              |
| 7          | A              |
| 8          | common cathode |
| 9          | F              |
| 10         | G              |

Table 5.3: SMA42056 7-segment LED pin configuration.

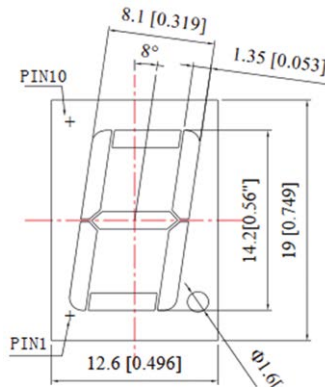


Figure 5.55: Pin numbering of the SMA42056 display.

### The aim

The aim of this project is to show how a 7-segment LED can be interfaced and used in Nucleo-L476RG projects.

### Block diagram

The block diagram of the project is shown in Figure 5.56.

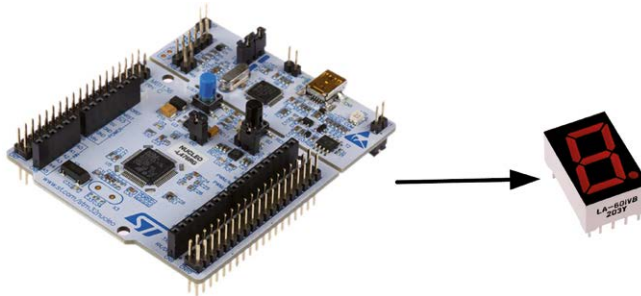


Figure 5.56: Block diagram of the project.

### Circuit diagram

The circuit diagram of the project is shown in Figure 5.57. The segments of the LED are connected to GPIO pins through 470-ohm current limiting resistors. The connection between the display segments and the GPIO pins are as follows:

| Segment | GPIO pin |
|---------|----------|
| a       | PC0      |
| b       | PC1      |
| c       | PC2      |
| d       | PC3      |
| e       | PC4      |
| f       | PC5      |
| g       | PC6      |

Before driving the display we have to know the relationship between the numbers to be displayed and the 7-segment pins that should be HIGH. This is shown below:

| Number to be Displayed | Segment to be ON |
|------------------------|------------------|
| 0                      | a b c d e f      |
| 1                      | b c              |
| 2                      | a b g e d        |
| 3                      | a b g c d        |
| 4                      | f g c b          |
| 5                      | a f g c d        |
| 6                      | a f g e d c      |
| 7                      | a b c            |
| 8                      | a b c d e f g    |
| 9                      | a b g f c d      |

The relationship between the numbers to be displayed and the data to be sent to port GPIOC in hexadecimal is derived from Table 5.4. In this table unused pins are set to zeroes. As an example, to display number 2 we have to send hexadecimal number 0x5B to GPIOC.

| Number | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Hexadecimal |
|--------|---|---|---|---|---|---|---|---|-------------|
| 0      | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0x3F        |
| 1      | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0x06        |
| 2      | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0x5B        |
| 3      | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0x4F        |
| 4      | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0x66        |
| 5      | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 0x6D        |
| 6      | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0x7D        |
| 7      | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0x07        |
| 8      | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0x7F        |
| 9      | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0x6F        |

Table 5.4: Number to be displayed and data to be sent to GPIOC.

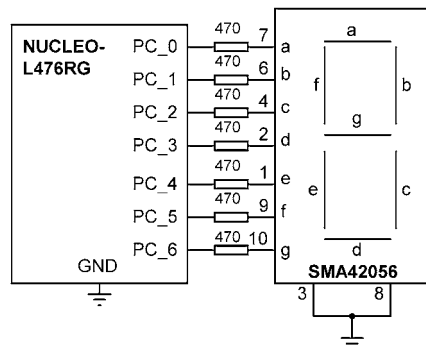


Figure 5.57: Circuit diagram of the project.

## The construction

The 7-segment LED and the current limiting resistors were mounted on a breadboard and then connections were made to Nucleo-L476RG connector using jumper wires.

## Program listing

Create a new program named **SEVENSEGMENT**, configure GPIO ports PC0 to PC6 as digital outputs and set the MCU clock to 80 MHz. The program listing is given in Figure 5.58 (comments have been removed for clarity).

As shown below, at the beginning of the program array **LEDS** is initialized to store the data to be sent to GPIOC to turn the appropriate segments ON to display the requested number:

```
int LEDS[] = {0x3F, /* 0 */
 0x06, /* 1 */
 0x5B, /* 2 */
 0x4F, /* 3 */
 0x66, /* 4 */
 0x6D, /* 5 */
 0x7D, /* 6 */
 0x07, /* 7 */
 0x7F, /* 8 */
 0x6F /* 9 */
};
```

The main program uses variable **Count** as the index for array **LEDS** where the data is sent to GPIOC using the GPIOC -> ODR statement. Variable **Count** is incremented by one and the process is repeated after one second delay. When **Count** reaches 10 it is reset to 0. The main program code is as follows:

```
while (1)
{
 GPIOC -> ODR = LEDS[Count];
 Count++;
 if(Count == 10)Count = 0;
 HAL_Delay(1000);
}

/* USER CODE BEGIN Header */
/**
 * *****
 * @file : main.c
 * @brief : Main program body
 * *****
 * @attention
 *
 * <h2><center>© Copyright (c) 2020 STMicroelectronics.
 * All rights reserved.</center></h2>
 */
```

```
*
* This software component is licensed by ST under BSD 3-Clause license,
* the «License»; You may not use this file except in compliance with the
* License. You may obtain a copy of the License at:
*
* opensource.org/licenses/BSD-3-Clause
*

*/
/* USER CODE END Header */
#include «main.h»

//
// Define the 7-segment LED connections
//
int LEDS[] = {0x3F, /* 0 */
 0x06, /* 1 */
 0x5B, /* 2 */
 0x4F, /* 3 */
 0x66, /* 4 */
 0x6D, /* 5 */
 0x7D, /* 6 */
 0x07, /* 7 */
 0x7F, /* 8 */
 0x6F /* 9 */
};

void SystemClock_Config(void);
static void MX_GPIO_Init(void);

int main(void)
{
 int Count = 0;
 Initialize Count
 HAL_Init();

 SystemClock_Config();

 MX_GPIO_Init();

 while (1)
 {
 GPIOC -> ODR = LEDS[Count];
 Count++;
 if(Count == 10)Count = 0;
 HAL_Delay(1000);
 }
}
```

```

}

void SystemClock_Config(void)
{
 RCC_OscInitTypeDef RCC_OscInitStruct = {0};
 RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

 RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
 RCC_OscInitStruct.HSISState = RCC_HSI_ON;
 RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
 RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
 RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
 RCC_OscInitStruct.PLL.PLLM = 2;
 RCC_OscInitStruct.PLL.PLLN = 20;
 RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV7;
 RCC_OscInitStruct.PLL.PLLQ = RCC_PLLQ_DIV2;
 RCC_OscInitStruct.PLL.PLLR = RCC_PLLR_DIV2;
 if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
 {
 Error_Handler();
 }

 RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
 |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
 RCC_ClkInitStruct.SYSClkSource = RCC_SYSCCLKSOURCE_PLLCLK;
 RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCCLK_DIV1;
 RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
 RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

 if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_4) != HAL_OK)
 {
 Error_Handler();
 }

 if (HAL_PWREx_ControlVoltageScaling(PWR_REGULATOR_VOLTAGE_SCALE1) != HAL_OK)
 {
 Error_Handler();
 }
}

static void MX_GPIO_Init(void)
{
 GPIO_InitTypeDef GPIO_InitStruct = {0};

 /* GPIO Ports Clock Enable */
 __HAL_RCC_GPIOC_CLK_ENABLE();

```

```

/*Configure GPIO pin Output Level */
HAL_GPIO_WritePin(GPIOC, GPIO_PIN_0|GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3
 |GPIO_PIN_4|GPIO_PIN_5|GPIO_PIN_6, GPIO_PIN_RESET);

/*Configure GPIO pins : PC0 PC1 PC2 PC3
 PC4 PC5 PC6 */
GPIO_InitStruct.Pin = GPIO_PIN_0|GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3
 |GPIO_PIN_4|GPIO_PIN_5|GPIO_PIN_6;
GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
GPIO_InitStruct.Pull = GPIO_NOPULL;
GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);

}

void Error_Handler(void)
{
}

#ifdef USE_FULL_ASSERT

void assert_failed(uint8_t *file, uint32_t line)
{
}

#endif

/***** (C) COPYRIGHT STMicroelectronics *****/

```

*Figure 5.58: Program: SEVENSEGMENT.*

Compile the program and make sure there are no errors. Upload the binary file **SEVEN-SEGMENT.bin** by dragging and dropping it to device NUCLEO-L476RG.

### 5.11 Project 10: Two-Digit Multiplexed 7-Segment LED

#### Description

This project is similar to the previous one but here two multiplexed digits are used instead of just one digit, and a fixed number is displayed. In this project, number 27 is displayed as an example. In multiplexed LED applications (see Figure 5.59) the LED segments of all the digits are tied together and the common pins of each digit are turned ON separately by the microcontroller. By displaying each digit for several milliseconds, the eye can not differentiate that the digits aren't ON all the time. This way we can multiplex any number of 7-segment displays together. For example, to display number 45, we have to send '4' to the first digit and enable its common pin. After a few milliseconds, the common pin of the first digit is disabled and the number '5' is sent to the second digit, next the common pin of the second digit is enabled. When this process is repeated continuously the user perceives both displays as ON continuously.

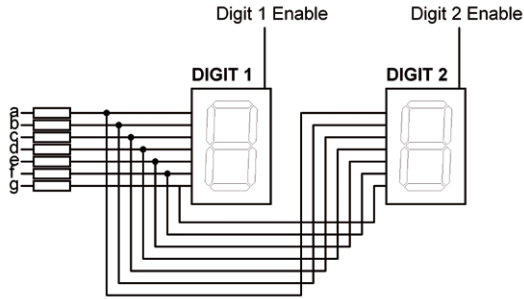


Figure 5.59: Two multiplexed 7-segment displays.

Some manufacturers provide multiplexed multi-digit displays in single packages. For example, we can purchase 2-, 4-, or 8-digit multiplexed displays in a single package. The display used in this project is the DC56-11EWA (see Figure 5.60) which is a red colour, 0.56-inch height, common-cathode two-digit display with 18 pins. The pin configuration of this display is shown in Table 5.5 and Figure 5.61. This display can be controlled from the microcontroller as follows:

- send the segment bit pattern for digit 1 to segments a to g;
- enable digit 1;
- wait for a few milliseconds;
- Disable digit 1;
- send the segment bit pattern for digit 2 to segments a to g;
- enable digit 2;
- wait for a few milliseconds;
- disable digit 2;
- repeat the above process continuously.

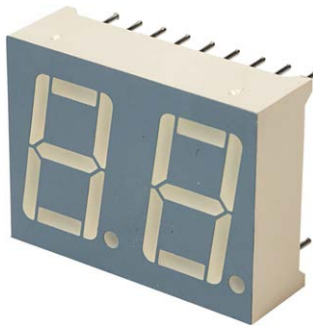


Figure 5.60: DC56-11EWA 2-digit display.



| Pin no | Segment         |
|--------|-----------------|
| 1,5    | E               |
| 2,6    | D               |
| 3,8    | C               |
| 14     | digit 1 Enable  |
| 17,7   | G               |
| 15,10  | B               |
| 16,11  | A               |
| 18,12  | F               |
| 13     | digit 2 Enable  |
| 4      | decimal Point1  |
| 9      | decimal Point 2 |

Table 5.5: Pin configuration of DC56-11EWA dual display.

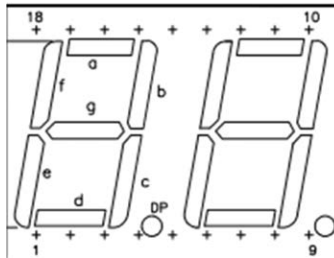


Figure 5.61: DC-11EWA pin configuration.

The segment configuration of the type DC56-11EWA display is shown in Figure 5.62. In a multiplexed display application the segment pins of corresponding segments are connected together. For example, pins 11 and 16 are connected as the common **a** segment. Similarly, pins 15 and 10 are connected as the common **b** segment, and so on.

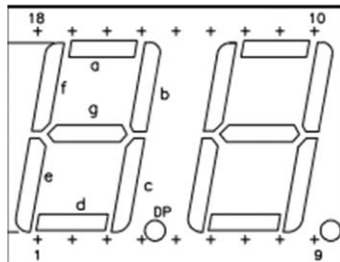


Figure 5.62: DC56-11EWA display segment configuration.

### The aim

The aim of this project is to show how a 2-digit 7-segment LED can be interfaced and used in Nucleo-L476RG projects.

### Block diagram

The block diagram of the project is shown in Figure 5.63.

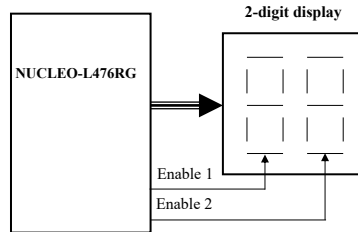


Figure 5.63: Block diagram of the project.

### Circuit diagram

The circuit diagram of the project is shown in Figure 5.64. The a–g pins of the two digits are joined and are then connected to the GPIOC pins as in the previous project. The digits are driven using NPN transistors (e.g. BC108/BC547). A digit is enabled when the base of its transistor is at logic 1, and is disabled when the base is cleared to logic 0. Digit 1 (10's digit) is driven from pin 7 of GPIOC (PC\_7), and digit 2 (1's digit) is driven from pin 8 of GPIOC (PC\_8).

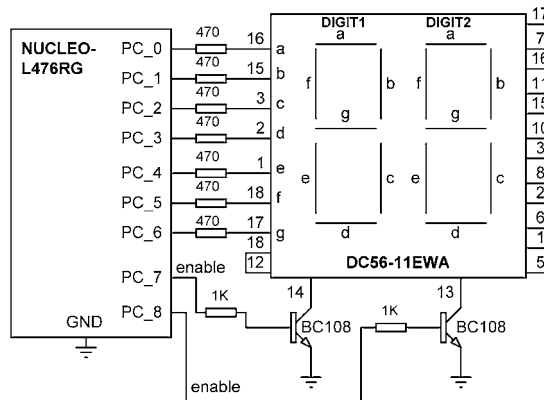


Figure 5.64: Circuit diagram of the project.

### The construction

The 7-segment LED and the current limiting resistors were mounted on a breadboard and then connections were made to Nucleo-L476RG connector using jumper wires.

### Project PDL

The operation of the program is shown in Figure 5.65.

```
BEGIN
 Create an array to store bit patterns
 Configure GPIOC as outputs
```

```

Initialize Count to 27
DO FOREVER
 Extract the MSD digit
 Extract the LSD digit
 Get bit pattern from the array for MSD digit
 Send bit pattern to GPIOC
 Enable digit 1
 Wait for 10ms
 Disable digit 1
 Get bit pattern from the array for LSD digit
 Send bit pattern to GPIOC
 Enable digit 2
 Wait for 10ms
 Disable digit 2
ENDDO
END

```

Figure 5.65: Operation of the program.

### Program listing

Create a program with the name **SEVENSEGMENT2**. Configure port pins PC0 to PC8 as digital outputs (see Figure 5.66).

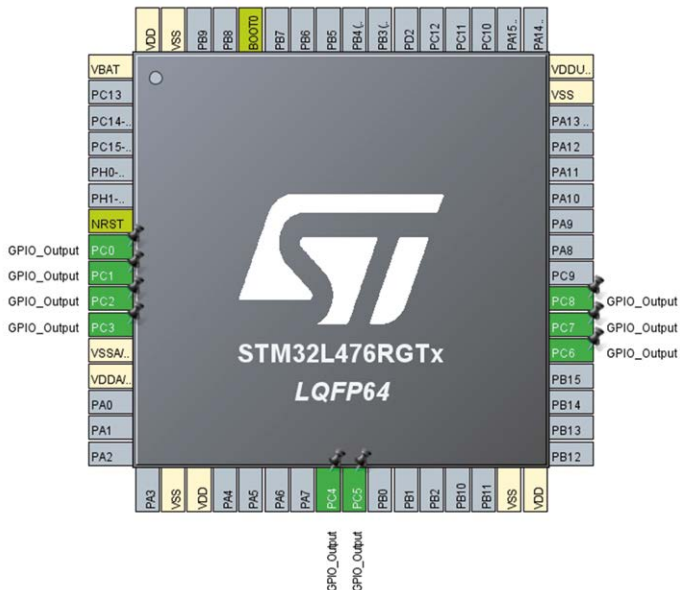


Figure 5.66: Configuring PC0–PC8 as digital outputs.

Figure 5.67 shows the program listing (comments are removed for clarity). At the beginning of the program variables **digit1** and **digit2** are assigned to pins 7 and 8 of PORTC respectively. Then, array **LEDS** is initialized to store the data to be sent to GPIOC to turn

the appropriate segments ON to display the requested number and variable **Count** is set to 27. The digit control bits are disabled before entering the program loop. Inside the program loop the MSD and LSD digits of **Count** are extracted and stored in variables **MSD** and **LSD** respectively. The MSD and LSD values are used as indexes in array **LEDS**. The **MSD** digit is first sent out and **digit1** is enabled for 10ms. Then the **LSD** digit is sent out and **digit2** is enabled for 10ms. The net result is that the display shows number 27.

```

/* USER CODE BEGIN Header */
/**
 * *****
 * @file : main.c
 * @brief : Main program body
 * *****
 * @attention
 *
 * <h2><center>© Copyright (c) 2020 STMicroelectronics.
 * All rights reserved.</center></h2>
 *
 * This software component is licensed by ST under BSD 3-Clause license,
 * the «License»; You may not use this file except in compliance with the
 * License. You may obtain a copy of the License at:
 *
 * opensource.org/licenses/BSD-3-Clause
 *
 * *****
 */
/* USER CODE END Header */
#include «main.h»

#define digit1 GPIO_PIN_7
#define digit2 GPIO_PIN_8

int LEDS[] = {0x3F, /* 0 */
 0x06, /* 1 */
 0x5B, /* 2 */
 0x4F, /* 3 */
 0x66, /* 4 */
 0x6D, /* 5 */
 0x7D, /* 6 */
 0x07, /* 7 */
 0x7F, /* 8 */
 0x6F /* 9 */
};

void SystemClock_Config(void);
static void MX_GPIO_Init(void);

```

```
int main(void)
{
 int MSD, LSD, Count = 27;
 HAL_Init();

 SystemClock_Config();

 MX_GPIO_Init();
 HAL_GPIO_WritePin(GPIOC, digit1, GPIO_PIN_RESET);
 HAL_GPIO_WritePin(GPIOC, digit2, GPIO_PIN_RESET);

 while (1)
 {
 MSD = Count / 10; // Get MSD
 LSD = Count % 10; // Get LSD
 GPIOC -> ODR = LEDS[MSD]; // Output MSD
 HAL_GPIO_WritePin(GPIOC, digit1, GPIO_PIN_SET); // Enable digit1
 HAL_Delay(10); // 10ms delay
 HAL_GPIO_WritePin(GPIOC, digit1, GPIO_PIN_RESET); // Disable
digit1

 GPIOC -> ODR = LEDS[LSD]; // Output LSD
 HAL_GPIO_WritePin(GPIOC, digit2, GPIO_PIN_SET); // Enable digit2
 HAL_Delay(10); // 10ms delay
 HAL_GPIO_WritePin(GPIOC, digit2, GPIO_PIN_RESET); // Disable digit2
 }
}

void SystemClock_Config(void)
{
 RCC_OscInitTypeDef RCC_OscInitStruct = {0};
 RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

 RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
 RCC_OscInitStruct.HSISState = RCC_HSI_ON;
 RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
 RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
 RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
 RCC_OscInitStruct.PLL.PLLM = 2;
 RCC_OscInitStruct.PLL.PLLN = 20;
 RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV7;
 RCC_OscInitStruct.PLL.PLLQ = RCC_PLLQ_DIV2;
 RCC_OscInitStruct.PLL.PLLR = RCC_PLLR_DIV2;
 if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
 {
```

```

 Error_Handler();
}

RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
 |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_4) != HAL_OK)
{
 Error_Handler();
}

if (HAL_PWREx_ControlVoltageScaling(PWR_REGULATOR_VOLTAGE_SCALE1) != HAL_OK)
{
 Error_Handler();
}
}

static void MX_GPIO_Init(void)
{
 GPIO_InitTypeDef GPIO_InitStruct = {0};

 /* GPIO Ports Clock Enable */
 __HAL_RCC_GPIOC_CLK_ENABLE();

 /*Configure GPIO pin Output Level */
 HAL_GPIO_WritePin(GPIOC, GPIO_PIN_0|GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3
 |GPIO_PIN_4|GPIO_PIN_5|GPIO_PIN_6|GPIO_PIN_7
 |GPIO_PIN_8, GPIO_PIN_RESET);

 /*Configure GPIO pins : PC0 PC1 PC2 PC3
 PC4 PC5 PC6 PC7
 PC8 */
 GPIO_InitStruct.Pin = GPIO_PIN_0|GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3
 |GPIO_PIN_4|GPIO_PIN_5|GPIO_PIN_6|GPIO_PIN_7
 |GPIO_PIN_8;
 GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
 GPIO_InitStruct.Pull = GPIO_NOPULL;
 GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
 HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);
}

void Error_Handler(void)

```

```
{
}

#ifdef USE_FULL_ASSERT

void assert_failed(uint8_t *file, uint32_t line)
{
}
#endif

/***** (C) COPYRIGHT STMicroelectronics *****/
```

*Figure 5.67: Program: SEVENSEGMENT2.*

## 5.12 Project 11: External interrupt to control an LED

### Description

In the last Project we have seen how to develop a two digit 7-segment display. The problem with multidigit 7-segment displays is that the display needs to be refreshed constantly and as a result of this, the MCU cannot perform any other functions while refreshing the display. There are 3 ways round this problem: 1. Using a multitasking kernel, 2. Using external interrupts to create the number to be displayed, 3. Using a timer interrupt to refresh the display in the background.

In this section we will learn how to generate external interrupts using the STM32CubeIDE programs. After learning this, we will see in the next Project how an external interrupt can be used to create numbers and display on a multidigit 7-segment display.

In this project the on-board button and the on-board LED are used such that pressing the button will generate an external interrupt which will toggle the state of the LED.

### Aim

The aim of this project is to show how an external interrupt can be generated and handled using the STM32CubeIDE.

### Block diagram

The block diagram of the project is as shown in Figure 5.34.

### Circuit diagram

The circuit diagram of the project is as shown in Figure 5.35, where the on-board button and on-board LED are used.

### Program listing

As described in Chapter 2, there are 16 external interrupt lines with 16 multiplexers and separate interrupt vector addresses that are connected with GPIO pins. These are named as EXTI0, EXTI1, etc up to EXTI15. GPIO pins with the same order are grouped together and are connected to an EXTI line. For example, EXTI2 is connected to PA2, PB2, PC2 and so on. This is why at any given instance we can have an external interrupt in only one of connected GPIO pins of that EXTI multiplexer. For example, when we need to use EXTI3,

we can use PA3, PB3, PC3, etc. but not PA2, PB2, etc. simultaneously. Thus, an entire GPIO port or port pins from different GPIO ports can be configured as external interrupts. We can also decide when to detect an external interrupt — on rising or falling edges, or both. The EXTI lines 0–4 are individually and directly connected to the NVIC interface while the remaining higher order EXTI lines are grouped into two — one ranging from EXTI 5 to 9 and the other from 10–15. The upper EXTI pins are thus not individually and directly connected to the NVIC. Owing to this, EXTI0–EXTI4 have separate unique interrupt vector addresses while the rest have two separate interrupt vector addresses.

In this project we will be using port pin PC13 (on-board button) and PA5 (on-board LED). Port pin PC13 is connected to interrupt line EXTI13, which is controlled by NVIC interface group EXTI15\_10.

The following steps are required to configure the MCU to accept interrupts on port pin PC\_13:

- configure the GPIO mode to accept rising (low-to-high) or falling (high-to-low) interrupts;
- enable the interrupt of the EXTI line associated with pin PC13, that is **EXTI15\_10\_IRQn**;
- define function **void EXTI15\_10\_IRQHandler(void)**, which is the ISR routine associated to the IRQ for the EXTI15\_10 line inside the vector table (lines 66:69).

In this project, we will toggle the state of the LED inside the ISR. The steps to develop the program are as follows:

- start STM32CubeIDE;
- create a new workspace;
- click to start a new STM32 project;
- give the name **EXTERNALINT** to the program;
- click on PA5 and configure it as digital output;
- click on PC13 and select GPIO\_EXTI13 as shown in Figure 5.68;



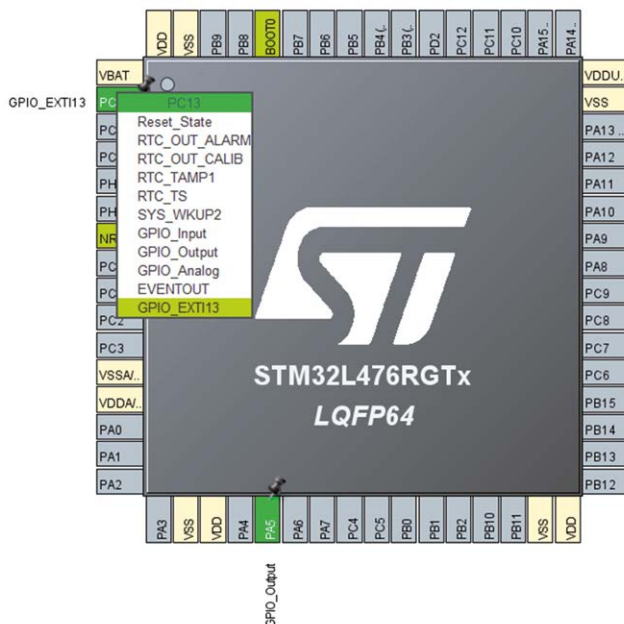


Figure 5.68: Configuring pins PA\_5 and PC\_13.

- configure the MCU clock to operate at 80 MHz;
- click **System Core**, followed by **GPIO** and then click on PC13 and set the **GPIO mode** to **External interrupt mode with falling edge trigger detection** as shown in Figure 5.69;

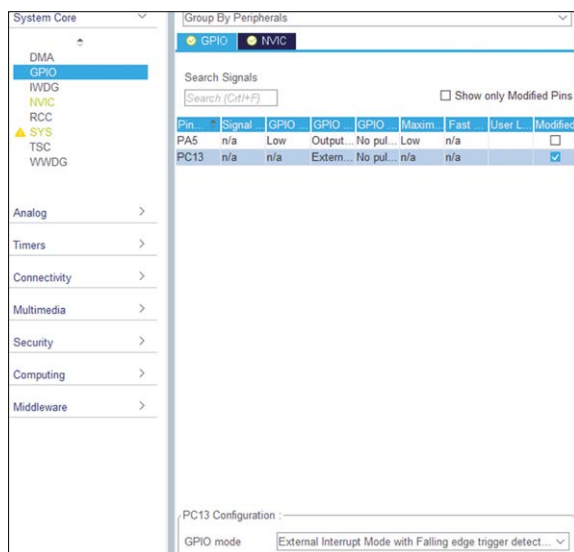


Figure 5.69 Select falling edge detection.

- click **NVIC** and click to enable **EXTI line[15:10]** and set the **Preemptio priority** and **Subpriority** to 0 as shown in Figure 5.70;

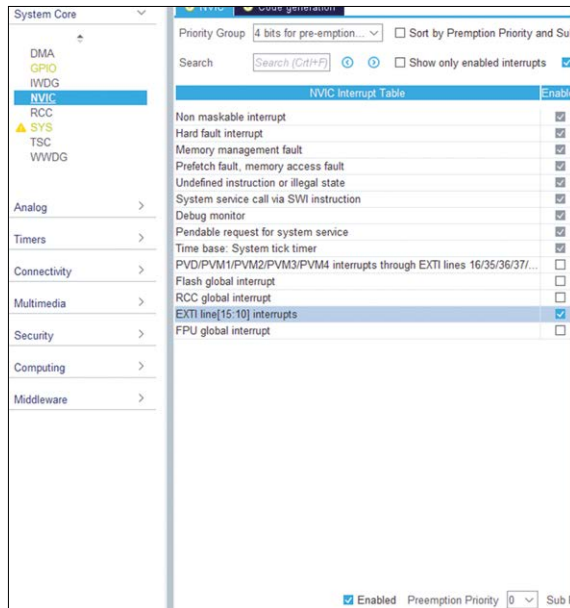


Figure 5.70: Enable EXTI line.

- click **File**, followed by **Save** and click **YES** to generate code;
- enter the following statement for the LED: `#define myled GPIO_PIN_5` ;
- notice the following additional statements at the end of function **MX\_GPIO\_Init(void)**, where the interrupt priority is set, and external interrupt is enabled on line EXTI15\_10:

```
/* EXTI interrupt init*/
HAL_NVIC_SetPriority(EXTI15_10_IRQn, 0, 0);
HAL_NVIC_EnableIRQ(EXTI15_10_IRQn);
```

It is interesting to notice that every time an external interrupt occurs, the function called **EXTI15\_10\_IRQHandler** in file **stm32l4xx\_it.c** under folder **src** is activated. The contents of this function is as follows (comments removed for clarity):

```
void EXTI15_10_IRQHandler(void)
{
 HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_13);
}
```

We should create the interrupt service routine function with the name **HAL\_GPIO\_EXTI\_Callback** and write our interrupt routine inside this function. Here, the only code we have inside the interrupt service routine is to toggle the state of the LED:

```
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
 HAL_GPIO_TogglePin(GPIOA, myled);
}
```

The main program loop has nothing useful to do in this program. Figure 5.71 shows the complete program listing of the project.

```
/* USER CODE BEGIN Header */
/**

 * @file : main.c
 * @brief : Main program body

 * @attention
 *
 * <h2><center>© Copyright (c) 2020 STMicroelectronics.
 * All rights reserved.</center></h2>
 *
 * This software component is licensed by ST under BSD 3-Clause license,
 * the «License»; You may not use this file except in compliance with the
 * License. You may obtain a copy of the License at:
 *
 * opensource.org/licenses/BSD-3-Clause
 *

 */
#include «main.h»

#define myled GPIO_PIN_5

void SystemClock_Config(void);
static void MX_GPIO_Init(void);

//
// Interrupt service routine, called when an external interrupt occurs
// i.e. when the button is pressed the LED is toggled by this function
//
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
 HAL_GPIO_TogglePin(GPIOA, myled);
}
```

```
//
// Start of main program loop. Notice that there is no code in the loop
//
int main(void)
{
 HAL_Init();

 SystemClock_Config();

 MX_GPIO_Init();

 while (1)
 {
 }
}

void SystemClock_Config(void)
{
 RCC_OscInitTypeDef RCC_OscInitStruct = {0};
 RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

 RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
 RCC_OscInitStruct.HSISState = RCC_HSI_ON;
 RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
 RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
 RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
 RCC_OscInitStruct.PLL.PLLM = 2;
 RCC_OscInitStruct.PLL.PLLN = 20;
 RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV7;
 RCC_OscInitStruct.PLL.PLLQ = RCC_PLLQ_DIV2;
 RCC_OscInitStruct.PLL.PLLR = RCC_PLLR_DIV2;
 if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
 {
 Error_Handler();
 }

 RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
 |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
 RCC_ClkInitStruct.SYSClkSource = RCC_SYSCCLKSOURCE_PLLCLK;
 RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCCLK_DIV1;
 RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
 RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

 if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_4) != HAL_OK)
 {
 Error_Handler();
 }
}
```

```
}

if (HAL_PWREx_ControlVoltageScaling(PWR_REGULATOR_VOLTAGE_SCALE1) != HAL_OK)
{
 Error_Handler();
}
}

static void MX_GPIO_Init(void)
{
 GPIO_InitTypeDef GPIO_InitStruct = {0};

 /* GPIO Ports Clock Enable */
 __HAL_RCC_GPIOC_CLK_ENABLE();
 __HAL_RCC_GPIOA_CLK_ENABLE();

 /*Configure GPIO pin Output Level */
 HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_RESET);

 /*Configure GPIO pin : PC13 */
 GPIO_InitStruct.Pin = GPIO_PIN_13;
 GPIO_InitStruct.Mode = GPIO_MODE_IT_FALLING;
 GPIO_InitStruct.Pull = GPIO_NOPULL;
 HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);

 /*Configure GPIO pin : PA5 */
 GPIO_InitStruct.Pin = GPIO_PIN_5;
 GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
 GPIO_InitStruct.Pull = GPIO_NOPULL;
 GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
 HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);

 /* EXTI interrupt init*/
 HAL_NVIC_SetPriority(EXTI15_10_IRQn, 0, 0);
 HAL_NVIC_EnableIRQ(EXTI15_10_IRQn);

}

void Error_Handler(void)
{
}

#ifdef USE_FULL_ASSERT

void assert_failed(uint8_t *file, uint32_t line)
{

```

```

}
#endif

/***** (C) COPYRIGHT STMicroelectronics *****/

```

*Figure 5.71: Program: EXTERNALINT.*

### Interrupt processing

The STM32 MCU generally has 7 interrupt handlers for GPIO pins as shown in Table 5.6. The table shows which IRQ to set for NVIC (first column) and function names to handle the interrupts (second column). Notice that only lines 0–4 have their own IRQ handlers. Lines 5–9 have the same interrupt handler. Also, lines 10–15 have the same interrupt handlers. If we are using multiple interrupts from lines 5–9 or from 10–15, then we have to check in software which line has actually caused the interrupt.

| IRQ            | Handler              | Description                                      |
|----------------|----------------------|--------------------------------------------------|
| EXTI0_IRQn     | EXTI0_IRQHandler     | Handler for GPIO pins connected to line 0        |
| EXTI1_IRQn     | EXTI1_IRQHandler     | Handler for GPIO pins connected to line 1        |
| EXTI2_IRQn     | EXTI2_IRQHandler     | Handler for GPIO pins connected to line 2        |
| EXTI3_IRQn     | EXTI3_IRQHandler     | Handler for GPIO pins connected to line 3        |
| EXTI4_IRQn     | EXTI4_IRQHandler     | Handler for GPIO pins connected to line 4        |
| EXTI9_5_IRQn   | EXTI9_5_IRQHandler   | Handler for GPIO pins connected to line 5 to 9   |
| EXTI15_10_IRQn | EXTI15_10_IRQHandler | Handler for GPIO pins connected to line 10 to 15 |

*Table 5.6: STM32 MCU interrupt handlers for GPIO.*

Interrupts are enabled using the function: **HAL\_NVIC\_EnableIRQ(IRQn\_Type IRQn)**. The corresponding function to disable an IRQ is: **HAL\_NVIC\_DisableIRQ(IRQn\_Type IRQn)**. Cortex-M automatically performs most of the work for interrupt processing.

An interrupt can be in one of three states:

- disabled or enabled;
- pending or not pending;
- active or inactive.

When an enabled interrupt occurs, it is put into the state of pending until the processor is ready to serve it. If there are no other interrupts currently being processed, then its pending state is cleared and the interrupt becomes active and is processed by the MCU.

Function **HAL\_NVIC\_GetPendingIRQ(IRQn\_Type IRQn)** can be called to see if an interrupt is pending but is not active. A 0 is returned if the IRQ is not pending, otherwise 1 is returned. The pending bit of an IRQ can be set by the function call **HAL\_NVIC\_SetPendingIRQ(IRQn\_Type IRQn)**. Setting the pending bit will cause the interrupt to be in a pending state where it can be active when the processor is not serving any other interrupts. The pending bit of an interrupt can be cleared using function **HAL\_NVIC\_ClearPendingIRQ(IRQn\_Type IRQn)**. Function **HAL\_NVIC\_GetActive(IRQn\_Type IRQn)** can be used to check if an interrupt is currently active.

The priority of the interrupts is defined through an 8-bit IPR register, which allows up to 255 different priority levels. The lower the priority level, the higher is the actual priority of the interrupt. In STM32 processors, only the 4 upper bits of the IPR register is used while its lower bits are all zeroes. This means that there are only 16 priority levels: 0x00, 0x10, 0x20, 0x30, 0x40, 0x50, 0x60, 0x70, 0x80, 0x90, 0xA0, 0xB0, 0xC0, 0xD0, 0xE0, 0xF0. Because a lower priority number has a higher priority, for example an IRQ with a priority 0x20 has a higher priority than an IRQ with a priority level equal to 0x80. The priority is an important feature of the MCU. If two interrupts occur at the same time, then the one with the higher priority will be serviced first by the MCU. Also, if the processor is servicing an interrupt, another interrupt with higher priority will suspend the current interrupt and the MCU will service the interrupt with higher priority. When the higher priority interrupt terminates, the execution will go back to the previous interrupt.

The IPR register is logically divided into two parts: a number of bits defining the **Preemptive** priority, and a number of bits defining the **Subpriority** priority. The **Preemptive** priority rules the pre-emption priorities between ISRs. The **Subpriority** determines which ISR will be executed first if there are multiple pending interrupts with the same **Preemptive** priorities. An interrupt with higher **Subpriority** (i.e. lower **Subpriority** number) will be given the service if multiple interrupts with equal **Preemptive** priorities are pending and a higher **Preemptive** interrupt is active, and then this higher **Preemptive** interrupt terminates. As an example, assume that there are three interrupt sources named **A**, **B**, and **C**, all having the 0 as their **Preemptive** priority. Also assume that interrupt source **B** has **Subpriority** 0 and interrupt source **C** has **Subpriority** 1. Now assume that interrupt **A** is active. When interrupt **B** comes in it is now set as pending. Also, when interrupt **C** comes in, it is also set as pending. Now, when interrupt **A** terminates, the next interrupt to become active is **B** since it has a higher **Subpriority** than **C** (i.e. lower **Subpriority** number).

As shown in Table 5.7, five priority groupings are defined where the **Preemptive** and **Subpriority** numbers can be specified.

| Priority Group       | No of Preemptive priority levels | No of Subpriority levels |
|----------------------|----------------------------------|--------------------------|
| NVIC_PRIORITYGROUP_0 | 0                                | 16                       |
| NVIC_PRIORITYGROUP_1 | 2                                | 8                        |
| NVIC_PRIORITYGROUP_2 | 4                                | 4                        |
| NVIC_PRIORITYGROUP_3 | 8                                | 2                        |
| NVIC_PRIORITYGROUP_4 | 16                               | 0                        |

Table 5.7: Priority groupings.

The interrupt priority can be set during the configuration phase as described in this project. Alternatively, we can set the interrupt priority using the function call **HAL\_NVIC\_SetPriority(IRQn\_Type IRQn, PreemptPriority, SubPriority)**. The PreemptPriority and SubPriority can be configured from 0 to 16 (these values are shifted to the 4 upper bits of the IPR register).

The priority grouping can be defined using the function **HAL\_NVIC\_SetPriorityGrouping(PriorityGroup)** where the **PriorityGroup** is one of the groups in Table 5.6.

The priority of an interrupt can be obtained by calling function **HAL\_NVIC\_GetPriority(IRQn\_Type IRQn, PriorityGroup, PreemptPrio, SubPriority)**. Also, the current priority grouping can be obtained using function: **HAL\_NVIC\_GetPriorityGrouping(void)**.

Interrupt latency is an important parameter we should know in time critical applications. This term refers to the number of clock cycles required for the MCU to respond to an interrupt request. Interrupt latency is usually measured by the number of clock cycles between the assertion of the interrupt request up to the point where the first instruction of the interrupt handler will be executed. The interrupt latency of the Cortex-M processors is very low. Cortex-M3 and Cortex-M4 processors both have 12 cycles of interrupt latency. These values assume that the memory system has zero wait state, the interrupt service is not blocked by another running interrupt, and the current executing instruction is not doing an unaligned transfer which can add one extra transfer cycle.

### 5.13 Project 12: Two-digit Interrupt-Driven 7-Segment Event Counter

#### Description

This project is similar to Project 9, but here the user button on the Nucleo-L476RG board gets configured to generate external interrupts when pressed, and then a count is incremented inside the interrupt service routine. This count is then displayed on the 2-digit 7-segment LED display. Pressing the button simulates the arrival of external events and therefore the program is a simple event counter.

#### The aim

The aim of this project is to show how an external interrupt can be used in a program with a multidigit 7-segment display.

#### Block diagram

The block diagram of the project is shown in Figure 5.72. User button on the board is used to generate external interrupts.

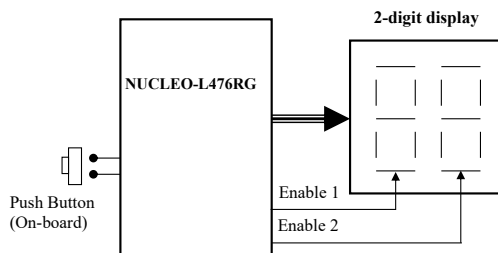


Figure 5.72: Block diagram of the project.

#### Circuit diagram

The circuit diagram of the project is as in Figure 5.64.

#### Program listing

The steps to write the program are as follows:



- 
- Pinout diagram of the STM32L476RGTx LQFP64 package. The diagram shows the chip with its 64 pins. A callout box highlights the PC13 pin, which is connected to the Reset\_State pin. The pinout is as follows:
- | Pin           | Function      |
|---------------|---------------|
| VDD           | Power Supply  |
| VSS           | Ground        |
| PB9           | GPIO Output   |
| PB8           | GPIO Output   |
| B00T0         | GPIO Output   |
| PB7           | GPIO Output   |
| PB6           | GPIO Output   |
| PB5           | GPIO Output   |
| PB4           | GPIO Output   |
| PB3           | GPIO Output   |
| PD2           | GPIO Output   |
| PC12          | GPIO Output   |
| PC11          | GPIO Output   |
| PC10          | GPIO Output   |
| PA15          | GPIO Output   |
| PA14          | GPIO Output   |
| VDDU          | Power Supply  |
| PA13          | GPIO Output   |
| PA12          | GPIO Output   |
| PA11          | GPIO Output   |
| PA10          | GPIO Output   |
| PA9           | GPIO Output   |
| PA8           | GPIO Output   |
| PC9           | GPIO Output   |
| PC8           | GPIO Output   |
| PC7           | GPIO Output   |
| PC6           | GPIO Output   |
| PB15          | GPIO Output   |
| PB14          | GPIO Output   |
| PB13          | GPIO Output   |
| PB12          | GPIO Output   |
| VDD           | Power Supply  |
| VSS           | Ground        |
| PA3           | GPIO Output   |
| PA4           | GPIO Output   |
| PA5           | GPIO Output   |
| PA6           | GPIO Output   |
| PA7           | GPIO Output   |
| PC4           | GPIO Output   |
| PC5           | GPIO Output   |
| PB0           | GPIO Output   |
| PB1           | GPIO Output   |
| PB2           | GPIO Output   |
| PB10          | GPIO Output   |
| PB11          | GPIO Output   |
| VSS           | Ground        |
| VDDA          | Power Supply  |
| VSSA          | Ground        |
| PA2           | GPIO Output   |
| PA1           | GPIO Output   |
| PA0           | GPIO Output   |
| PC3           | GPIO Output   |
| PC2           | GPIO Output   |
| PC1           | GPIO Output   |
| PC6           | GPIO Output   |
| NR5           | GPIO Output   |
| PH1           | GPIO Output   |
| PH0           | GPIO Output   |
| PC1           | GPIO Output   |
| PC1           | GPIO Output   |
| PC1           | GPIO Output   |
| Reset_State   | Reset State   |
| RTC_OUT_ALARM | RTC Output    |
| RTC_OUT_CALIB | RTC Output    |
| RTC_TAMP1     | RTC Output    |
| RTC_TS        | RTC Output    |
| SYS_WKUP2     | System Wakeup |
| GPIO_Input    | GPIO Input    |
| GPIO_Output   | GPIO Output   |
| GPIO_Analog   | GPIO Analog   |
| EVENTOUT      | Event Output  |
| GPIO_EXTI13   | GPIO Output   |

- Configure the MCU clock to operate at 80 MHz.
- Click **System Core**, followed by **GPIO** and then click on PC\_13 and set the **GPIO mode** to **External interrupt mode with falling edge trigger detection** as described in Project 10.
- Click **NVIC** and click to enable **EXTI line[15:10]** and set the **Preemption priority** and **Subpriority** to 0 as described in Project 10.
- Click **File**, followed by **Save** and click **YES** to generate code.
- Enter the following statements just after the **#include "main.h"** statement. These statements define the digit connections and also the bit patterns to be sent to the 7-segment display in order to display the required numbers:

```
#define digit1 GPIO_PIN_7
#define digit2 GPIO_PIN_8

int LEDS[] = {0x3F, /* 0 */
 0x06, /* 1 */
 0x5B, /* 2 */
 0x4F, /* 3 */
```

```

 0x66, /* 4 */
 0x6D, /* 5 */
 0x7D, /* 6 */
 0x07, /* 7 */
 0x7F, /* 8 */
 0x6F, /* 9 */
 };

 int Count = 0;

```

- The interrupt service routine is named **HAL\_GPIO\_EXTI\_Callback**. The program jumps to this function as soon as an interrupt is detected on pin PC13, i.e. as soon as the button is pressed. This function increments variable **Count** by one:

```

Void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
 Count++;
 if(Count == 100)Count = 0;
}

```

- Inside the main program loop, the MSD and LSD of variable **Count** are extracted. The MSD digit is enabled and displayed first, followed by the LSD digit. Notice that the MSD digit is blanked if the value of **Count** is less than 10, i.e. if MSD is zero.

Compile the program after making sure that the compiler is in Release mode. If there are no errors, then drag and drop the binary file **INT7SEGMENT.bin** to device NODE\_L476RG. Press the on-board user button and you should see the count incrementing on the display. Figure 5.74 shows the complete program listing.

```

/* USER CODE BEGIN Header */
/**
 *
 * *****
 * @file : main.c
 * @brief : Main program body
 *
 * *****
 * @attention
 *
 * <h2><center>© Copyright (c) 2020 STMicroelectronics.
 * All rights reserved.</center></h2>
 *
 * This software component is licensed by ST under BSD 3-Clause license,
 * the «License»; You may not use this file except in compliance with the
 * License. You may obtain a copy of the License at:
 *
 * opensource.org/licenses/BSD-3-Clause
 *
 */

```

```

*/
#include «main.h»

#define digit1 GPIO_PIN_7
#define digit2 GPIO_PIN_8

int LEDS[] = {0x3F, /* 0 */
 0x06, /* 1 */
 0x5B, /* 2 */
 0x4F, /* 3 */
 0x66, /* 4 */
 0x6D, /* 5 */
 0x7D, /* 6 */
 0x07, /* 7 */
 0x7F, /* 8 */
 0x6F /* 9 */
};

int Count = 0;

void SystemClock_Config(void);
static void MX_GPIO_Init(void);

//
// This is the Interrupt Service Routine. The program jumps here when the
// button is pressed
//
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
 Count++;
 if(Count == 100)Count = 0;
}

//
// Start of main program loop
//
int main(void)
{
 int MSD, LSD;
 HAL_Init();

 SystemClock_Config();

 MX_GPIO_Init();
 HAL_GPIO_WritePin(GPIOC, digit1, GPIO_PIN_RESET); // Clear digit1

```

```

HAL_GPIO_WritePin(GPIOC, digit2, GPIO_PIN_RESET); // Clear digit2

while (1)
{
 MSD = Count / 10; // Extract MSD digit
 LSD = Count % 10; // Extract LSD digit
 if(MSD != 0) // If MSD is non zero
 {
 GPIOC -> ODR = LEDS[MSD]; // Send MSD to display
 HAL_GPIO_WritePin(GPIOC,digit1,GPIO_PIN_SET); // Enable digit1
 HAL_Delay(10); // Wait 10ms
 HAL_GPIO_WritePin(GPIOC,digit1,GPIO_PIN_RESET); // Disable digit1
 }

 GPIOC -> ODR = LEDS[LSD]; // Send LSD digit
 HAL_GPIO_WritePin(GPIOC,digit2,GPIO_PIN_SET); // Enable digit2
 HAL_Delay(10); // Wait 10ms
 HAL_GPIO_WritePin(GPIOC,digit2,GPIO_PIN_RESET); // Disable digit2
}
}

void SystemClock_Config(void)
{
 RCC_OscInitTypeDef RCC_OscInitStruct = {0};
 RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

 RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
 RCC_OscInitStruct.HSISState = RCC_HSI_ON;
 RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
 RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
 RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
 RCC_OscInitStruct.PLL.PLLM = 2;
 RCC_OscInitStruct.PLL.PLLN = 20;
 RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV7;
 RCC_OscInitStruct.PLL.PLLQ = RCC_PLLQ_DIV2;
 RCC_OscInitStruct.PLL.PLLR = RCC_PLLR_DIV2;
 if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
 {
 Error_Handler();
 }

 RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
 |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
 RCC_ClkInitStruct.SYSClkSource = RCC_SYSCCLKSOURCE_PLLCLK;
 RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCCLK_DIV1;
 RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
}

```

```
RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_4) != HAL_OK)
{
 Error_Handler();
}

if (HAL_PWREx_ControlVoltageScaling(PWR_REGULATOR_VOLTAGE_SCALE1) != HAL_OK)
{
 Error_Handler();
}
}

static void MX_GPIO_Init(void)
{
 GPIO_InitTypeDef GPIO_InitStruct = {0};

 /* GPIO Ports Clock Enable */
 __HAL_RCC_GPIOC_CLK_ENABLE();

 /*Configure GPIO pin Output Level */
 HAL_GPIO_WritePin(GPIOC, GPIO_PIN_0|GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3
 |GPIO_PIN_4|GPIO_PIN_5|GPIO_PIN_6|GPIO_PIN_7
 |GPIO_PIN_8, GPIO_PIN_RESET);

 /*Configure GPIO pin : PC13 */
 GPIO_InitStruct.Pin = GPIO_PIN_13;
 GPIO_InitStruct.Mode = GPIO_MODE_IT_FALLING;
 GPIO_InitStruct.Pull = GPIO_NOPULL;
 HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);

 /*Configure GPIO pins : PC0 PC1 PC2 PC3
 PC4 PC5 PC6 PC7
 PC8 */
 GPIO_InitStruct.Pin = GPIO_PIN_0|GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3
 |GPIO_PIN_4|GPIO_PIN_5|GPIO_PIN_6|GPIO_PIN_7
 |GPIO_PIN_8;
 GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
 GPIO_InitStruct.Pull = GPIO_NOPULL;
 GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
 HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);

 /* EXTI interrupt init*/
 HAL_NVIC_SetPriority(EXTI15_10_IRQn, 0, 0);
 HAL_NVIC_EnableIRQ(EXTI15_10_IRQn);
}
```

```

}

void Error_Handler(void)
{
}

#ifdef USE_FULL_ASSERT

void assert_failed(uint8_t *file, uint32_t line)
{
}
#endif

/***** (C) COPYRIGHT STMicroelectronics *****/

```

*Figure 5.74: Program: INT7SEGMENT.*

### 5.14 Project 13: Four-Digit 7-Segment LED Display

#### Description

This project is similar to Project 9, but here a 4-digit, 7-segment display is developed. In this project, number '1234' is displayed as an example. In 4-digit multiplexed LED applications the LED segments of all the digits are tied together and the common pins of each digit are turned ON separately by the microcontroller. By displaying each digit for several milliseconds, the eye can not differentiate that the digits are not ON all the time. This way we can multiplex any number of 7-segment displays together. For example, to display number '1234', we have to send '1' to the first digit and enable its common pin. After a few milliseconds, the common pin of the first digit is disabled and the number '2' is sent to the second digit and the common pin of the second digit is enabled. Then, the number '3' is sent and its digit is enabled. Finally, '4' gets sent and its digit is enabled. When this process is repeated continuously the user sees as if both displays are ON continuously.

The display used in this project is the DC56-11EWA as in Project 9, which is a red colour, 0.56-inch height, common-cathode, two-digit display with 18 pins. Two such displays are used to make a 4-digit display.

#### The aim

The aim of this project is to show how a 4-digit 7-segment LED can be interfaced and used in Nucleo-L476RG projects.

#### Block diagram

The block diagram of the project is shown in Figure 5.75.

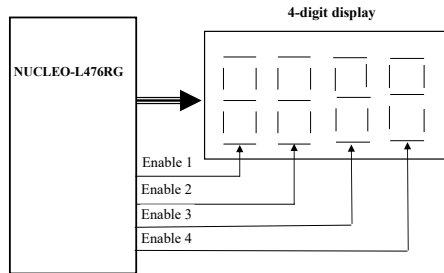


Figure 5.75: Block diagram of the project.

### Circuit diagram

The circuit diagram of the project is shown in Figure 5.76. The a–g pins of the four digits are connected together and are then connected to the GPIOC pins as in Project 9. The digits are driven using NPN transistors (e.g. BC108/BC547). A digit is enabled when the base of its transistor is at logic 1, and is disabled when the base is cleared to logic 0. Digit 1 (1000's digit) is driven from PC7, digit 2 (100's digit) from PC8, digit 3 (10's) from PC9, and digit 4 (1's) from PC10.

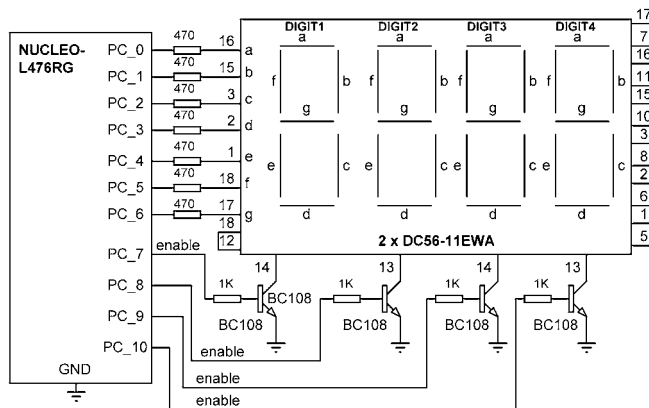


Figure 5.76: Circuit diagram of the project.

### The construction

The 7-segment LED and the current limiting resistors were mounted on a breadboard and then connections were made to Nucleo-L476RG connector using jumper wires.

### Program listing

The steps to write the program are as follows:

- Start STM32CubeIDE.
- Create a new workspace.
- Click to start a new STM32 project.
- Give the name **SEVEN4** to the program.
- Configure PC0 to PC10 as digital output (Figure 5.77).
- Configure the MCU clock for 80 MHz.

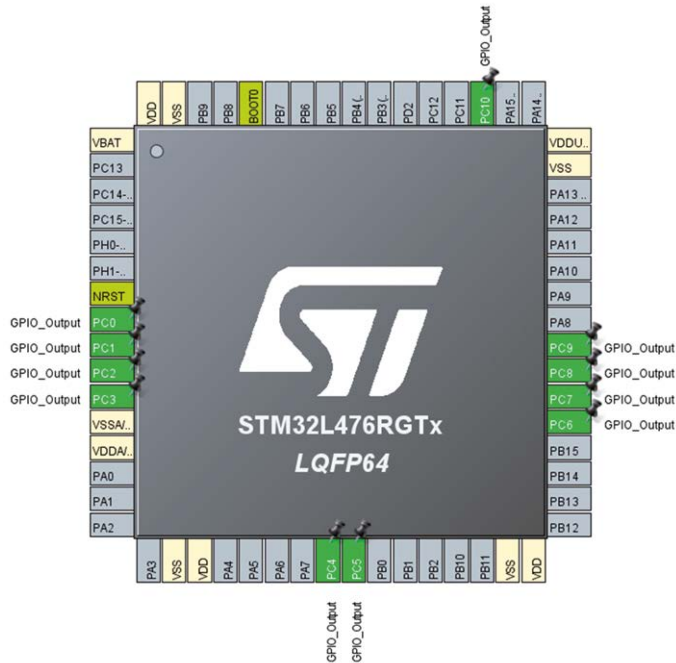


Figure 5.77: Configure the GPIO.

- Enter the following statements just after the **#include "main.h"** statement. These statements define the digit connections and also the bit patterns to be sent to the 7-segment display in order to display the required numbers:

```
#define digit1 GPIO_PIN_7
#define digit2 GPIO_PIN_8
#define digit3 GPIO_PIN_9
#define digit4 GPIO_PIN_10

int LEDS[] = {0x3F, /* 0 */
 0x06, /* 1 */
 0x5B, /* 2 */
 0x4F, /* 3 */
 0x66, /* 4 */
 0x6D, /* 5 */
 0x7D, /* 6 */
 0x07, /* 7 */
 0x7F, /* 8 */
 0x6F, /* 9 */
 };
```

- At the beginning of the program all four digits are disabled:



```

HAL_GPIO_WritePin(GPIOC, digit1, GPIO_PIN_RESET); // Disable digit1
HAL_GPIO_WritePin(GPIOC, digit2, GPIO_PIN_RESET); // Disable digit2
HAL_GPIO_WritePin(GPIOC, digit3, GPIO_PIN_RESET); // Disable digit3
HAL_GPIO_WritePin(GPIOC, digit4, GPIO_PIN_RESET); // Disable digit4

```

- The digits of the number to be displayed are stored in variables MSD, MID2, MID1, and LSD:

```

MSD = Count / 1000; // Get MSD
m = Count % 1000;
MID2 = m / 100; // Get MID2
n = m % 100;
MID1 = n / 10; // Get MID1
LSD = n % 10; // Get LSD

```

- The digits are enabled for 5ms each. For example, digit1 is enabled as follows:

```

GPIOC -> ODR = LEDS[MSD]; // Output MSD
HAL_GPIO_WritePin(GPIOC, digit1, GPIO_PIN_SET); // Enable digit1
HAL_Delay(5); // 5ms delay
HAL_GPIO_WritePin(GPIOC, digit1, GPIO_PIN_RESET); // Disable digit1

```

Figure 5.78 shows the complete program listing. Compile the program and make sure that there are no errors. Drag and drop the binary file **SEVEN4.bin** to device NUCLEO\_L476RG.

```

/* USER CODE BEGIN Header */
/**
 * *****
 * @file : main.c
 * @brief : Main program body
 * *****
 * @attention
 *
 * <h2><center>© Copyright (c) 2020 STMicroelectronics.
 * All rights reserved.</center></h2>
 *
 * This software component is licensed by ST under BSD 3-Clause license,
 * the «License»; You may not use this file except in compliance with the
 * License. You may obtain a copy of the License at:
 *
 * opensource.org/licenses/BSD-3-Clause
 *
 * *****
 */
#include «main.h»

#define digit1 GPIO_PIN_7

```

```

#define digit2 GPIO_PIN_8
#define digit3 GPIO_PIN_9
#define digit4 GPIO_PIN_10

int LEDS[] = {0x3F, /* 0 */
 0x06, /* 1 */
 0x5B, /* 2 */
 0x4F, /* 3 */
 0x66, /* 4 */
 0x6D, /* 5 */
 0x7D, /* 6 */
 0x07, /* 7 */
 0x7F, /* 8 */
 0x6F /* 9 */
 };

void SystemClock_Config(void);
static void MX_GPIO_Init(void);

int main(void)
{
 int LSD, MID1, MID2, MSD, m, n, Count = 1234;
 HAL_Init();

 SystemClock_Config();

 MX_GPIO_Init();

 HAL_GPIO_WritePin(GPIOC, digit1, GPIO_PIN_RESET); // Disable digit1
 HAL_GPIO_WritePin(GPIOC, digit2, GPIO_PIN_RESET); // Disable digit2
 HAL_GPIO_WritePin(GPIOC, digit3, GPIO_PIN_RESET); // Disable digit3
 HAL_GPIO_WritePin(GPIOC, digit4, GPIO_PIN_RESET); // Disable digit4

 while (1)
 {
 MSD = Count / 1000; // Get MSD
 m = Count % 1000;
 MID2 = m / 100; // Get MID2
 n = m % 100;
 MID1 = n / 10; // Get MID1
 LSD = n % 10; // Get LSD

 GPIOC -> ODR = LEDS[MSD]; // Output MSD
 HAL_GPIO_WritePin(GPIOC, digit1, GPIO_PIN_SET); // Enable digit1
 HAL_Delay(5); // 5ms delay
 }
}

```

```
 HAL_GPIO_WritePin(GPIOC, digit1, GPIO_PIN_RESET); // Disable digit1

 GPIOC -> ODR = LEDS[MID2]; // Output MID2
 HAL_GPIO_WritePin(GPIOC, digit2, GPIO_PIN_SET); // Enable digit2
 HAL_Delay(5); // 5ms delay
 HAL_GPIO_WritePin(GPIOC, digit2, GPIO_PIN_RESET); // Disable digit2

 GPIOC -> ODR = LEDS[MID1]; // Output MID1
 HAL_GPIO_WritePin(GPIOC, digit3, GPIO_PIN_SET); // Enable digit3
 HAL_Delay(5); // 5ms delay
 HAL_GPIO_WritePin(GPIOC, digit3, GPIO_PIN_RESET); // Disable digit3

 GPIOC -> ODR = LEDS[LSD]; // Output LSD
 HAL_GPIO_WritePin(GPIOC, digit4, GPIO_PIN_SET); // Enable digit4
 HAL_Delay(5); // 5ms delay
 HAL_GPIO_WritePin(GPIOC, digit4, GPIO_PIN_RESET); // Disable digit4
 }
}
```

```
void SystemClock_Config(void)
{
 RCC_OscInitTypeDef RCC_OscInitStruct = {0};
 RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

 RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
 RCC_OscInitStruct.HSISState = RCC_HSI_ON;
 RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
 RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
 RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
 RCC_OscInitStruct.PLL.PLLM = 2;
 RCC_OscInitStruct.PLL.PLLN = 20;
 RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV7;
 RCC_OscInitStruct.PLL.PLLQ = RCC_PLLQ_DIV2;
 RCC_OscInitStruct.PLL.PLLR = RCC_PLLR_DIV2;
 if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
 {
 Error_Handler();
 }

 RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
 |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
 RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
 RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
 RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
 RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;
```

```
if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_4) != HAL_OK)
{
 Error_Handler();
}

if (HAL_PWREx_ControlVoltageScaling(PWR_REGULATOR_VOLTAGE_SCALE1) != HAL_OK)
{
 Error_Handler();
}
}

static void MX_GPIO_Init(void)
{
 GPIO_InitTypeDef GPIO_InitStruct = {0};

 /* GPIO Ports Clock Enable */
 __HAL_RCC_GPIOC_CLK_ENABLE();

 /*Configure GPIO pin Output Level */
 HAL_GPIO_WritePin(GPIOC, GPIO_PIN_0|GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3
 |GPIO_PIN_4|GPIO_PIN_5|GPIO_PIN_6|GPIO_PIN_7
 |GPIO_PIN_8|GPIO_PIN_9|GPIO_PIN_10, GPIO_PIN_RESET);

 /*Configure GPIO pins : PC0 PC1 PC2 PC3
 PC4 PC5 PC6 PC7
 PC8 PC9 PC10 */
 GPIO_InitStruct.Pin = GPIO_PIN_0|GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3
 |GPIO_PIN_4|GPIO_PIN_5|GPIO_PIN_6|GPIO_PIN_7
 |GPIO_PIN_8|GPIO_PIN_9|GPIO_PIN_10;
 GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
 GPIO_InitStruct.Pull = GPIO_NOPULL;
 GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
 HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);

}

void Error_Handler(void)
{
}

#ifdef USE_FULL_ASSERT

void assert_failed(uint8_t *file, uint32_t line)
{
}

#endif
```

```
#endif
```

```
/***** (C) COPYRIGHT STMicroelectronics *****/
```

Figure 5.78: Program: SEVEN4.

### 5.15 Project 14: Interrupt-Based Up/Down Counter with Four-Digit 7-Segment LED Display

#### Description

In this project two external buttons named **UP** and **DOWN** are connected to the Nucleo board. Additionally, a 4-digit 7-segment LED is connected to the board. The buttons are configured as external interrupts. Pressing **UP** creates an interrupt and increments the count by 1. Similarly, pressing **DOWN** creates an interrupt and decrements the count by 1.

#### The aim

The aim of this project is to show how multiple external interrupts can be used in a program.

#### Block diagram

The block diagram of the project is shown in Figure 5.79. The user button on the board is used to generate external interrupts.

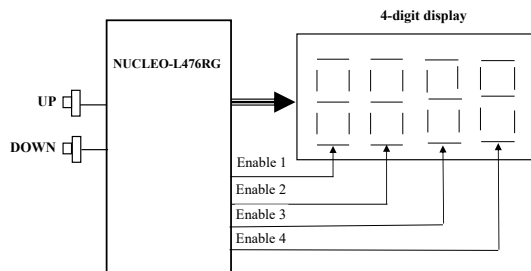


Figure 5.79: Block diagram of the project.

#### Circuit diagram

The circuit diagram of the project is shown in Figure 5.80. The UP and DOWN buttons are connected to PA0 and PA1 pins of the development board respectively. The state of these buttons are normally at logic 1 and go to logic 0 when the buttons are pressed. The 4-digit 7-segment LED is connected to as in the previous projects.

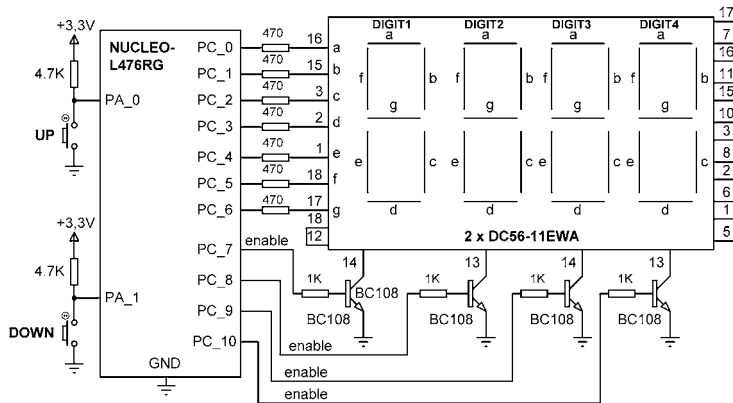


Figure 5.80: Circuit diagram of the project.

### Program listing

In this program port pins PA0 and PA1 get configured as external interrupts, and PC0 to PC10, as digital outputs.

The steps are:

- Start STM32CubeIDE.
- Create a new workspace.
- Click to start a new STM32 project.
- Give the name **UPDOWNINT** to the program.
- Configure PC0 to PC10 as digital output.
- Click on PA0 and select GPIO\_EXTI0.
- Click on PA1 and select GPIO\_EXTI1 (see Figure 5.81).

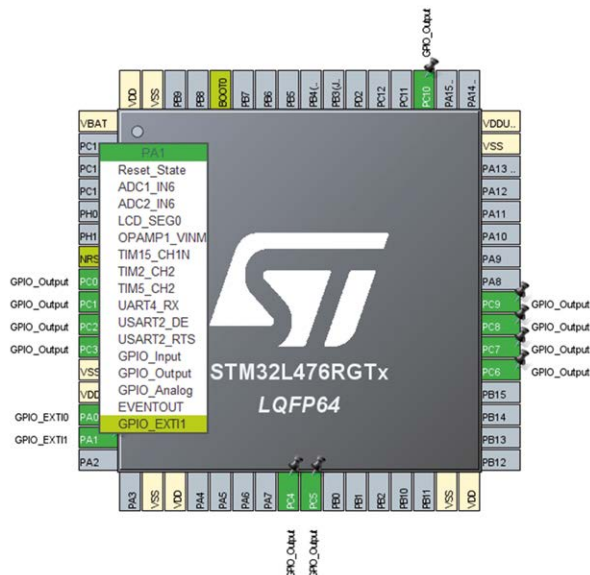


Figure 5.81: Configure the GPIO.

- Configure the MCU clock to operate at 80 MHz.
- Click **System Core**, followed by **GPIO** and then click on PA0 and set the **GPIO mode** to **External interrupt mode with falling edge trigger detection** (Figure 5.82).

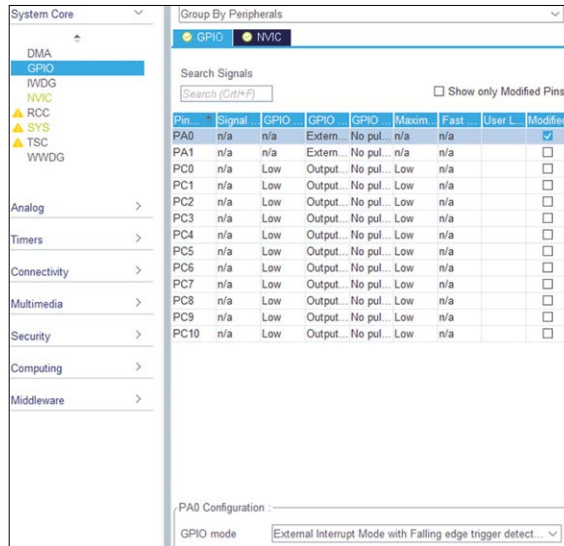


Figure 5.82: Configure falling-edge detection.

- Do the same for PA1.
- Click **NVIC** and click to enable **EXTI line0 interrupt** and **EXTI line1 interrupt**. Set the **Preemption priority** and **Subpriority** to 0 as shown in Figure 5.83.

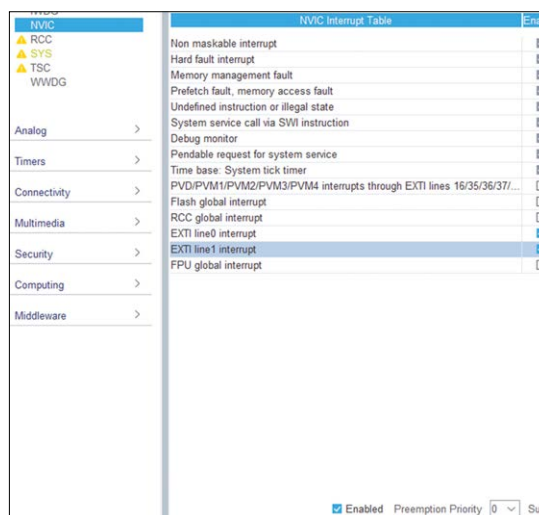


Figure 5.83: Configure the interrupt lines.

- Click **File**, followed by **Save** and click **YES** to generate code.
- Click **Core**, followed by **Src** and double click **main.c** to display the main program.

Notice that external interrupts are initialized with the following statements. The first two statements set the interrupt priority of EXTI0 to 0 and then enable interrupts on line EXTI0. Similarly, second two statements set the interrupt priority and enable interrupts for line EXTI1:

```
/* EXTI interrupt init*/
HAL_NVIC_SetPriority(EXTI0_IRQn, 0, 0);
HAL_NVIC_EnableIRQ(EXTI0_IRQn);

HAL_NVIC_SetPriority(EXTI1_IRQn, 0, 0);
HAL_NVIC_EnableIRQ(EXTI1_IRQn);
```

Every time an external interrupt occurs, the function called EXTI0\_IRQHandler or EXTI1\_IRQHandler in file stm32l4xx\_itc.c under folder Src is activated. The contents of this file relevant to external interrupts on lines EXTI0 and EXTI1 are as follows (comments have been removed for clarity):

```
void EXTI0_IRQHandler(void)
{
 HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_0);
}

void EXTI1_IRQHandler(void)
{
 HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_1);
}
```

We should create the interrupt service routine function with the name **HAL\_GPIO\_EXTI\_Callback** and increment or decrement variable Count depending on whether the UP or DOWN buttons are pressed.

At the beginning of the program, UP and DOWN buttons are defined, the four digits and LED bit patterns are defined as in the earlier projects, and variable **Count** is initialized to 0:

```
#define UP GPIO_PIN_0
#define DOWN GPIO_PIN_1

#define digit1 GPIO_PIN_7
#define digit2 GPIO_PIN_8
#define digit3 GPIO_PIN_9
#define digit4 GPIO_PIN_10

int LEDS[] = {0x3F, /* 0 */
```



```

 0x06, /* 1 */
 0x5B, /* 2 */
 0x4F, /* 3 */
 0x66, /* 4 */
 0x6D, /* 5 */
 0x7D, /* 6 */
 0x07, /* 7 */
 0x7F, /* 8 */
 0x6F /* 9 */
 };

 int Count = 0;

```

All four digits are then disabled just before the program loop. The contents of the main program loop are same as in the previous project which displays the number in variable **Count**. Notice that in this program, 4ms delay is used for refreshing the display digits which gives more stable display.

The interrupt service routine checks which pin caused the interrupt and then increments or decrements the value of variable **Count**:

```

void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
 if(GPIO_Pin == UP)
 {
 Count++;
 if(Count > 9999)Count = 0;
 }
 else if(GPIO_Pin == DOWN)
 {
 Count--;
 if(Count < 0)Count = 0;
 }
}

```

Figure 5.84 shows the complete program listing. Compile the program in Release mode and make sure there are no errors. Drag and drop the binary file **UPDOWNINT.bin** to device NUCLEO\_L476RG.

```

/* USER CODE BEGIN Header */
/**
 *
 * *****
 * @file : main.c
 * @brief : Main program body
 * *****
 * @attention
 *
 * <h2><center>© Copyright (c) 2020 STMicroelectronics.

```

```

* All rights reserved.</center></h2>
*
* This software component is licensed by ST under BSD 3-Clause license,
* the «License»; You may not use this file except in compliance with the
* License. You may obtain a copy of the License at:
*
* opensource.org/licenses/BSD-3-Clause
*

*/
#include «main.h»

#define UP GPIO_PIN_0
#define DOWN GPIO_PIN_1

#define digit1 GPIO_PIN_7
#define digit2 GPIO_PIN_8
#define digit3 GPIO_PIN_9
#define digit4 GPIO_PIN_10

int LEDS[] = {0x3F, /* 0 */
 0x06, /* 1 */
 0x5B, /* 2 */
 0x4F, /* 3 */
 0x66, /* 4 */
 0x6D, /* 5 */
 0x7D, /* 6 */
 0x07, /* 7 */
 0x7F, /* 8 */
 0x6F /* 9 */
};

int Count = 0;

void SystemClock_Config(void);
static void MX_GPIO_Init(void);

//
// This is the interrupt service routine. The program jumps here when an external
// interrupt occurs i.e. when the UP or DOWN buttons are pressed. Pressing the UP
// button increments Count, pressing the DOWN button decrements Count
//
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
 if(GPIO_Pin == UP)
 {
 Count++;
 if(Count > 9999)Count = 0;
 }
}

```

```
 }
 else if(GPIO_Pin == DOWN)
 {
 Count--;
 if(Count < 0)Count = 0;
 }
}

//
// Start of main program
//
int main(void)
{
 int MSD, MID2, MID1, LSD, m, n;
 HAL_Init();

 SystemClock_Config();

 MX_GPIO_Init();

 HAL_GPIO_WritePin(GPIOC, digit1, GPIO_PIN_RESET); // Disable digit1
 HAL_GPIO_WritePin(GPIOC, digit2, GPIO_PIN_RESET); // Disable digit2
 HAL_GPIO_WritePin(GPIOC, digit3, GPIO_PIN_RESET); // Disable digit3
 HAL_GPIO_WritePin(GPIOC, digit4, GPIO_PIN_RESET); // Disable digit4

 while (1)
 {
 MSD = Count / 1000; // Get MSD
 m = Count % 1000;
 MID2 = m / 100; // Get MID2
 n = m % 100;
 MID1 = n / 10; // Get MID1
 LSD = n % 10; // Get LSD

 GPIOC -> ODR = LEDS[MSD]; // Output MSD
 HAL_GPIO_WritePin(GPIOC, digit1, GPIO_PIN_SET); // Enable digit1
 HAL_Delay(4); // 4ms delay
 HAL_GPIO_WritePin(GPIOC, digit1, GPIO_PIN_RESET); // Disable digit1

 GPIOC -> ODR = LEDS[MID2]; // Output MID2
 HAL_GPIO_WritePin(GPIOC, digit2, GPIO_PIN_SET); // Enable digit2
 HAL_Delay(4); // 4ms delay
 HAL_GPIO_WritePin(GPIOC, digit2, GPIO_PIN_RESET); // Disable digit2

 GPIOC -> ODR = LEDS[MID1]; // Output MID1
```

```

 HAL_GPIO_WritePin(GPIOC, digit3, GPIO_PIN_SET); // Enable digit3
 HAL_Delay(4); // 4ms delay
 HAL_GPIO_WritePin(GPIOC, digit3, GPIO_PIN_RESET); // Disable digit3

 GPIOC -> ODR = LEDS[LSD]; // Output LSD
 HAL_GPIO_WritePin(GPIOC, digit4, GPIO_PIN_SET); // Enable digit4
 HAL_Delay(4); // 4ms delay
 HAL_GPIO_WritePin(GPIOC, digit4, GPIO_PIN_RESET); // Disable digit4
 }
}

```

```

void SystemClock_Config(void)
{
 RCC_OscInitTypeDef RCC_OscInitStruct = {0};
 RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

 RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
 RCC_OscInitStruct.HSISState = RCC_HSI_ON;
 RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
 RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
 RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
 RCC_OscInitStruct.PLL.PLLM = 2;
 RCC_OscInitStruct.PLL.PLLN = 20;
 RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV7;
 RCC_OscInitStruct.PLL.PLLQ = RCC_PLLQ_DIV2;
 RCC_OscInitStruct.PLL.PLLR = RCC_PLLR_DIV2;
 if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
 {
 Error_Handler();
 }

 RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
 |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
 RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
 RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
 RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
 RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

 if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_4) != HAL_OK)
 {
 Error_Handler();
 }

 if (HAL_PWREx_ControlVoltageScaling(PWR_REGULATOR_VOLTAGE_SCALE1) != HAL_OK)
 {

```

```
 Error_Handler();
}
}

static void MX_GPIO_Init(void)
{
 GPIO_InitTypeDef GPIO_InitStruct = {0};

 /* GPIO Ports Clock Enable */
 __HAL_RCC_GPIOC_CLK_ENABLE();
 __HAL_RCC_GPIOA_CLK_ENABLE();

 /*Configure GPIO pin Output Level */
 HAL_GPIO_WritePin(GPIOC, GPIO_PIN_0|GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3
 |GPIO_PIN_4|GPIO_PIN_5|GPIO_PIN_6|GPIO_PIN_7
 |GPIO_PIN_8|GPIO_PIN_9|GPIO_PIN_10, GPIO_PIN_RESET);

 /*Configure GPIO pins : PC0 PC1 PC2 PC3
 PC4 PC5 PC6 PC7
 PC8 PC9 PC10 */
 GPIO_InitStruct.Pin = GPIO_PIN_0|GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3
 |GPIO_PIN_4|GPIO_PIN_5|GPIO_PIN_6|GPIO_PIN_7
 |GPIO_PIN_8|GPIO_PIN_9|GPIO_PIN_10;
 GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
 GPIO_InitStruct.Pull = GPIO_NOPULL;
 GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
 HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);

 /*Configure GPIO pins : PA0 PA1 */
 GPIO_InitStruct.Pin = GPIO_PIN_0|GPIO_PIN_1;
 GPIO_InitStruct.Mode = GPIO_MODE_IT_FALLING;
 GPIO_InitStruct.Pull = GPIO_NOPULL;
 HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);

 /* EXTI interrupt init*/
 HAL_NVIC_SetPriority(EXTI0_IRQn, 0, 0);
 HAL_NVIC_EnableIRQ(EXTI0_IRQn);

 HAL_NVIC_SetPriority(EXTI1_IRQn, 0, 0);
 HAL_NVIC_EnableIRQ(EXTI1_IRQn);

}

void Error_Handler(void)
{
}
}
```

```

#ifdef USE_FULL_ASSERT

void assert_failed(uint8_t *file, uint32_t line)
{
}

#endif

/***** (C) COPYRIGHT STMicroelectronics *****/

```

Figure 5.84: Program: UPDOWNINT.

### Modified program

In the program in Figure 5.84, the interrupt service routine was defined by function **HAL\_GPIO\_EXTI\_Callback**. It is also possible to use the interrupt handler functions in file **stm32l4xx\_it.c**. This way, we do not have to include the **Callback** function in the main program. The code for the interrupt service routines inside file **stm32l4xx\_it.c** are shown below:

```

void EXTI0_IRQHandler(void)
{
 Count++;
 if(Count > 9999)Count = 0;
 HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_0);
}

void EXTI1_IRQHandler(void)
{
 Count--;
 if(Count < 0)Count = 0;
 HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_1);
}

```

Because variable **Count** is shared by the main program and by file **stm32l4xx\_it.c**, we have to define **Count** as an external variable in the main program as follows:

```
extern int Count = 0;
```

Also, variable **Count** must be defined at the beginning of file **stm32l4xx\_it.c** as follows:

```
int Count;
```

## 5.16 Project 15: Multiple External Interrupts Sharing the Same Interrupt Line

### Description

As we have seen in Table 5.6, external interrupts from GPIO ports 5-9 share the same interrupt line and the same interrupt handler function. Similarly, ports 10-15 share the same interrupt line and the same interrupt handler function.

In this project we have two LEDs connected to GPIO ports PA13 and PA14. Two buttons are used as external interrupts where the buttons share the same interrupt lines. Here, PA6 and PA7 are used as the button inputs which share the interrupt handler `EXTI9_5_IRQHandler`. In this project, pressing the button on the PA6 line toggles the LED at PA13. Similarly, pressing the button on PA7 toggles the LED on PA14.

### Aim

The aim of this project is to show how multiple interrupts on the same line can be handled by the MCU.

### Block Diagram

Figure 5.85 shows the block diagram of the project.

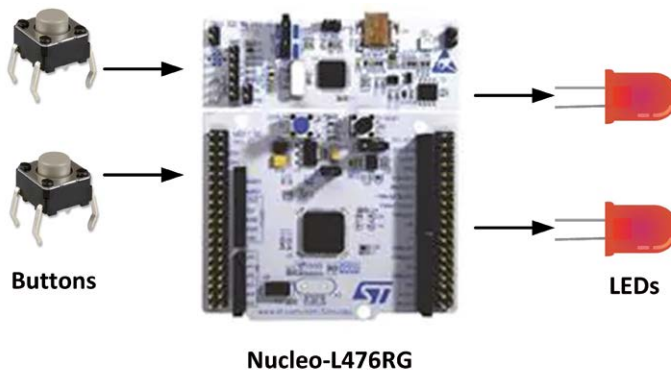


Figure 5.85: Block diagram of the project.

### Circuit diagram

The circuit diagram of the project is shown in Figure 5.86

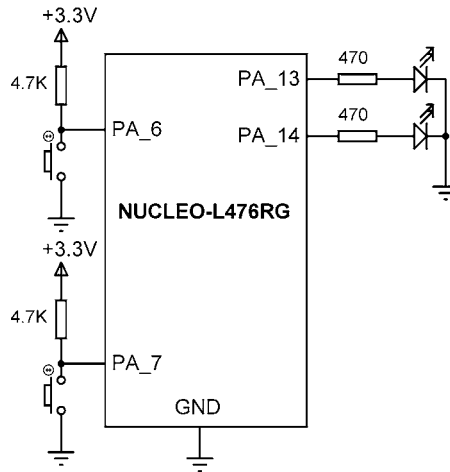


Figure 5.86: Circuit diagram of the project.

### Program listing

The steps are:

- Start STM32CubeIDE.
- Create a new workspace.
- Click to start a new STM32 project.
- Give the name **MULTIINT** to the program.
- Configure PA13 and PA14 as digital output.
- Click on PA6 and select GPIO\_EXTI6.
- Click on PA7 and select GPIO\_EXTI7 (see Figure 5.87).

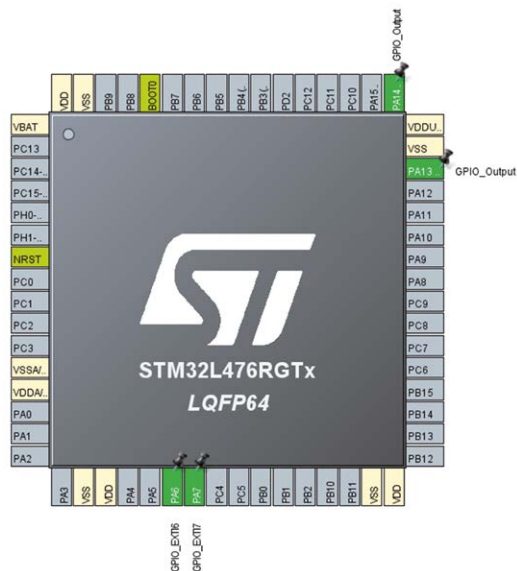


Figure 5.87: Configure the GPIO.



- Configure the MCU clock to operate at 80 MHz.
- Click **System Core**, followed by **GPIO** and then click on PA6 and set the **GPIO mode** to **External interrupt mode with falling edge trigger detection**
- Do the same for PA7.
- Click **NVIC** and click to enable **EXTI line0[9:5] interrupts**. Set the **Preemption priority** and **Subpriority** to 0.
- Click **File**, followed by **Save** and click **YES** to generate code.
- Click **Core**, followed by **Src** and double click **main.c** to display the main program.

Notice that external interrupts are initialized with the following statements. The first statement sets the interrupt priority of EXTI0 to 0. The second statement enables interrupts on line EXTI9\_5.

```
/* EXTI interrupt init*/
HAL_NVIC_SetPriority(EXTI9_5_IRQn, 0, 0);
HAL_NVIC_EnableIRQ(EXTI9_5_IRQn);
```

Every time an external interrupt occurs at port pins PA\_6 or PA\_7, the function called **EXTI9\_5\_IRQHandler** in file **stm32l4xx\_itc.c** under folder **Src** is activated. The contents of this file relevant to external interrupts is as follows (comments have been removed for clarity):

```
void EXTI9_5_IRQHandler(void)
{
 HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_6);
 HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_7);
}
```

The interrupt service routine function is called **HAL\_GPIO\_EXTI\_Callback** and it has the following contents. If **button1** is pressed then LED1 is toggled, if **button2** is pressed then LED2 is toggled:

```
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
 if(GPIO_Pin == button1) HAL_GPIO_TogglePin(GPIOA, LED1);
 if(GPIO_Pin == button2) HAL_GPIO_TogglePin(GPIOA, LED2);
}
```

Compile the program making sure that the Release mode is set. Drag and drop the binary file **MULTIINT.bin** to device Nucleo\_L476RG. Figure 5.88 shows the complete program listing (comments are removed for clarity).

```

/* USER CODE BEGIN Header */
/**

 * @file : main.c
 * @brief : Main program body

 * @attention
 *
 * <h2><center>© Copyright (c) 2020 STMicroelectronics.
 * All rights reserved.</center></h2>
 *
 * This software component is licensed by ST under BSD 3-Clause license,
 * the «License»; You may not use this file except in compliance with the
 * License. You may obtain a copy of the License at:
 *
 * opensource.org/licenses/BSD-3-Clause
 *

 */
#include «main.h»

#define button1 GPIO_PIN_6
#define button2 GPIO_PIN_7
#define LED1 GPIO_PIN_13
#define LED2 GPIO_PIN_14

void SystemClock_Config(void);
static void MX_GPIO_Init(void);

//
// This is the interrupt service routine. The program jumps here when either of the
// two external buttons are pressed
//
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
 if(GPIO_Pin == button1) HAL_GPIO_TogglePin(GPIOA, LED1);
 if(GPIO_Pin == button2) HAL_GPIO_TogglePin(GPIOA, LED2);
}

//
// Start of main program
//
int main(void)
{
 HAL_Init();

```

```
SystemClock_Config();

MX_GPIO_Init();

while (1)
{
}

}

void SystemClock_Config(void)
{
 RCC_OscInitTypeDef RCC_OscInitStruct = {0};
 RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

 RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
 RCC_OscInitStruct.HSISState = RCC_HSI_ON;
 RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
 RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
 RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
 RCC_OscInitStruct.PLL.PLLM = 2;
 RCC_OscInitStruct.PLL.PLLN = 20;
 RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV7;
 RCC_OscInitStruct.PLL.PLLQ = RCC_PLLQ_DIV2;
 RCC_OscInitStruct.PLL.PLLR = RCC_PLLR_DIV2;
 if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
 {
 Error_Handler();
 }

 RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
 |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
 RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
 RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
 RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
 RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

 if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_4) != HAL_OK)
 {
 Error_Handler();
 }

 if (HAL_PWREx_ControlVoltageScaling(PWR_REGULATOR_VOLTAGE_SCALE1) != HAL_OK)
 {
 Error_Handler();
 }
}
```

```

}

static void MX_GPIO_Init(void)
{
 GPIO_InitTypeDef GPIO_InitStruct = {0};

 /* GPIO Ports Clock Enable */
 __HAL_RCC_GPIOA_CLK_ENABLE();

 /*Configure GPIO pin Output Level */
 HAL_GPIO_WritePin(GPIOA, GPIO_PIN_13|GPIO_PIN_14, GPIO_PIN_RESET);

 /*Configure GPIO pins : PA6 PA7 */
 GPIO_InitStruct.Pin = GPIO_PIN_6|GPIO_PIN_7;
 GPIO_InitStruct.Mode = GPIO_MODE_IT_FALLING;
 GPIO_InitStruct.Pull = GPIO_NOPULL;
 HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);

 /*Configure GPIO pins : PA13 PA14 */
 GPIO_InitStruct.Pin = GPIO_PIN_13|GPIO_PIN_14;
 GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
 GPIO_InitStruct.Pull = GPIO_NOPULL;
 GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
 HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);

 /* EXTI interrupt init*/
 HAL_NVIC_SetPriority(EXTI9_5_IRQn, 0, 0);
 HAL_NVIC_EnableIRQ(EXTI9_5_IRQn);
}

void Error_Handler(void)
{
}

#ifdef USE_FULL_ASSERT

void assert_failed(uint8_t *file, uint32_t line)
{
}

#endif

/***** (C) COPYRIGHT STMicroelectronics *****/

```

*Figure 5.88: Program: MULTIINT.*

### 5.17 Summary

In this Chapter we have learned how to use the STM32CubeIDE to create a program, to compile it, and then to upload the binary file to the development board.

In the next Chapter we will be looking at the timers and develop several projects using the timers.

## CHAPTER 6 • Timers

### 6.1 Overview

Timers are important components of all MCUs. A timer is basically a free-running counter that counts pulses from a clock source. Because the period of the clock source is known, the elapsed time is known accurately as this is related to the number of clock pulses applied to the counter. The clock source for a timer is normally derived from the MCU clock. But because the MCU clock may be very fast, prescalers are used to divide and slow down the clock frequency before it is applied to the timer/counter. A counter can count up or down depending on its design and configuration. For example, a 16-bit up counter can count up to 65,535 before it overflows and returns to 0. During this overflow it can be configured to trigger an interrupt, known as timer interrupt. Similarly, a down counter can count down to zero and then trigger an interrupt.

Timers have many applications in time-based projects, including, but not limited to, the following:

- measuring the frequency of an incoming digital signal;
- generating pulse-width-modulated (PWM) signals to control heavy loads;
- generating interrupts at regular intervals;
- generating pulses with required frequency.

### 6.2 STM32 timers

STM32 timers can be grouped into several categories. The features of these categories are summarized below.

**Basic:** these are the simplest form of timers, having no input or output pins. These are 16-bit timers, generally used as clock sources in digital-to-analogue (DAC) and direct-memory-access (DMA) applications.

**General-purpose:** these are 16-bit or 32-bit timers, having both input and output pins. These timers can do all functions of Basic timers. These timers can have up to four programmable input/output channels which may be configured one or two channels, or as one or two channels with complementary outputs. The complementary outputs have dead-time generator that can provide independent timebase.

**Advanced:** these timers have all the features of general-purpose timers with additional features for heavy load control (e.g. motor control) and digital power conversion applications. These timers provide three complimentary outputs with an emergency shut-down input.

**High resolution:** these timers are only available on some members of the STM32 family. These timers allow to generate digital signals with high accuracy timings (e.g. PWM or phase shifted pulses). There are 10 high resolution outputs, configured as 6 sub-timers, 1 master and 5 slaves. The timers also feature 5 fault inputs for protection and 10 inputs to handle external events (e.g. current limitation, zero voltage switching etc).

**Low power:** these timers are used in low-power applications that can operate in most STM32 modes, except in Standby mode, also with no internal clock source. These timers can also wake up the MCU from low-power modes.

The STM32L476 processor includes multiple timers. Using the timers one can generate waveforms, react to external events, measure the parameters of incoming time varying signals and so on. The timers work in the background, thus off-loading the CPU from repetitive and time-critical tasks. The timers (except the low-power timer) are all based on the same architecture and they are scalable in terms of their resolution (16 or 32-bit), number of inputs (1 to 9), or their features (PWM, up/down counting, DMA). Timer channels can be configured as either input or output and multiple timers can be linked with each other and synchronized. The timers can also be used to provide timing signals to other modules inside the processor, such as the ADC.

The following timers are included on the STM32L476 processor:

- 2×16-bit advanced control timers (TIM1 and TIM8);
- 2× 32-bit general-purpose timers (TIM2 and TIM5);
- 5× 16-bit general-purpose timers (TIM3, TIM4, TIM15, TIM16, TIM17);
- 2× 16-bit basic timer (TIM6 and TIM7);
- 2× low-power 16-bit timers (LPTIM1 and LPTIM2);
- 2× watchdog timers;
- 1× SysTick timer.

TIM1 and TIM8 are 16-bit advanced control timers with up/down capabilities. These timers can also be used as general-purpose timers with 16-bit prescalers. These timers have complementary PWM outputs with programmable dead-times and 0% to 100% duty cycles so that they can be used in power control (e.g. motor control) applications with input capture and output compare functions.

Timers TIM2, TIM3, TIM4, TIM15, TIM16 and Tim17 are general-purpose timers. TIM2 and TIM5 are 32-bits wide up/down timers and have auto-reload functions with 32-bit prescalers. TIM3 and TIM4 are 16-bit up/down timers with auto-reload functions with 16-bit prescalers. All these timers have PWM or one-pulse outputs and input capture/output compare functions.

TIM15, TIM16, and TIM17 are general-purpose 16-bit timers with 16-bit auto-reload up-counters and 16-bit prescalers. These timers can be used for input capture/output compare, PWM or one-pulse outputs.

TIM6 and TIM7 are basic timers which are used for DAC trigger generation.

LPTIM1 and LPTIM2 are 16-bit low-power timers with 16-bit auto-reload registers. They have independent selectable clocks and are able to wake up the system from Stop mode. They have 16-bit compare registers and can output PWM or one-shot output pulses, programmable digital glitch filters and encoder modes (LPTIM1 only).

Figure 6.1 shows the internal structure of a general-purpose timer having 4 channels.

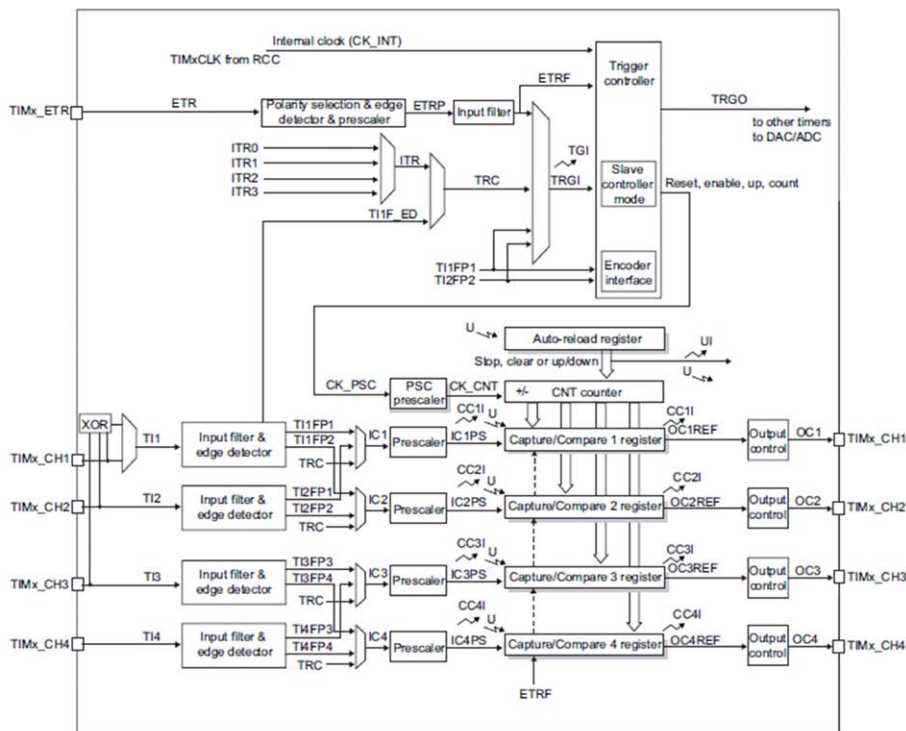


Figure 6.1: General-purpose timer (RM0351 Reference Manual, STMicroelectronics).

At the top of the diagram APB clock is selected as source and the internal clock CK\_INT feeds a prescaler (PSC). The output of the prescaler feeds a counter (CNT). The contents of the counter are compared to the auto-reload register and when they match an interrupt is triggered.

A general-purpose timer can be clocked from a variety of sources, including:

- the internal clock provided by the RCC;
- internal trigger input 0 to 3 (ITR0, ITR1, ITR2 and ITR3);
- external input channel pins;
  - pin 1: TI1FP1 or TI1F\_ED;
  - pin 2: TI2FP2;
- external ETR pins (ETR1 or ETR2).

In this Chapter we are only interested in using the internal clock.

### 6.3 Setting a timer

Figure 6.2 shows the registers used for setting a timer. The clock then enters the prescaler which can have a value between 1 and 65535. This derived clock drives timer register which is auto-reloaded from the auto-reload register after it counts down to 0.



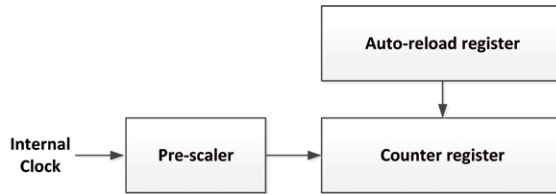


Figure 6.2: Timer operation.

The following equation can be used to determine the values to be loaded to generate interrupts at specified times:

$$I = \frac{(P+1)(A+1)}{f}$$

or,

$$A = \frac{f \times I}{(P+1)} - 1$$

where

$A$  is the value to be loaded into the auto-reload register;

$I$  is the required interrupt time in seconds;

$f$  is the clock period in Hz;

$P$  is the prescaler value (1 to 65535).

For example, with 80 MHz clock, for 1-second timer interrupts, assuming  $P = 39999$ , we have:

$$A = \frac{80,000,000 \times 1}{(39999+1)} - 1 = 1999$$

A project is given below which will use a timer to flash the LED every second.

## 6.4 Project 1: Timer Interrupt to Flash LED Every Second

### Description

In this project the on-board LED (at port PA5) is flashed every second using timer interrupts.

### The aim

The aim of this project is to show how timer interrupts can be used in a program.

### Program listing

In this program we will be using timer **TIM2** which is a general-purpose 32-bit timer. The steps are given below.

- Start STM32CubeIDE as before.
- Select the board type as Nucleo-L476RG.
- Give the name **TMRLED** to the program.

- Configure PA5 as digital output.
- Configure the MCU clock as 80 MHz and make sure that the APB1 clock is set to 80 MHz.
- Click **Timers**, then click TIM2, and select the **Clock Source** as **Internal clock** (Figure 6.3).

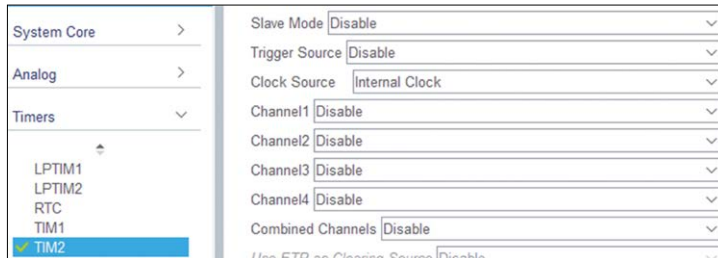


Figure 6.3: Select Internal Clock for TIM2.

- Under **Configuration**, click on **Prescaler (PSC – 16-bit value)** and make sure it is set to **Decimal** and change its value to 39999.
- Change the **Counter Mode** to **Down**.
- Click on **Counter Period (Auto reload register)** and make sure it is set to **Decimal**. Change its value to 1999.
- Click **Internal Clock Division** and set it to 2. Notice that this is not clock division, but it relates to the filter at the input sampler.
- Click **auto-reload preload** and click to **Enable** it.
- Click **System Core** followed by **NVIC**. Click to enable **TIM2 global interrupt**.
- Click **File**, followed by **Save** and click **YES** to generate code.

You should now open the generated code. Notice that the following code is included for the timer interrupt (comments are removed for clarity):

The following timer related code is included in the program:

```
TIM_HandleTypeDef htim2;
```

You should add the following code just before the **while** loop in the main program to enable timer interrupts:

```
HAL_TIM_Base_Start_IT(&htim2); // Enable the timer
```

Also, the timer interrupt service routine should be added before the main program as shown below. Notice that the LED control code to toggle the LED is inside the timer interrupt service routine.

```
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
 HAL_GPIO_TogglePin(GPIOA, LED);
}
```

Notice that the following code is added automatically to file **stm32l4xx\_it.c** for the timer interrupt (comments are removed):

```
void TIM2_IRQHandler(void)
{
 HAL_TIM_IRQHandler(&htim2);
}
```

Compile the program in **Release** mode and drag and drop the binary file **TMRLED.bin** to device NUCLEO\_L476RG. Figure 6.4 shows the complete program listing (comments are removed for clarity).

```
/* USER CODE BEGIN Header */
/**

 * @file : main.c
 * @brief : Main program body

 * @attention
 *
 * <h2><center>© Copyright (c) 2020 STMicroelectronics.
 * All rights reserved.</center></h2>
 *
 * This software component is licensed by ST under BSD 3-Clause license,
 * the «License»; You may not use this file except in compliance with the
 * License. You may obtain a copy of the License at:
 *
 * opensource.org/licenses/BSD-3-Clause
 *

 */
#include «main.h»
#define LED GPIO_PIN_5

TIM_HandleTypeDef htim2;

void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_TIM2_Init(void);

//
// This is the timer interrupt service routine. The program jumps here every second
//
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
 HAL_GPIO_TogglePin(GPIOA, LED);
}
```

```

int main(void)
{
 HAL_Init();

 SystemClock_Config();

 MX_GPIO_Init();
 MX_TIM2_Init();
 HAL_TIM_Base_Start_IT(&htim2); // Enable the timer

 while (1)
 {
 }
}

void SystemClock_Config(void)
{
 RCC_OscInitTypeDef RCC_OscInitStruct = {0};
 RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

 RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
 RCC_OscInitStruct.HSISState = RCC_HSI_ON;
 RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
 RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
 RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
 RCC_OscInitStruct.PLL.PLLM = 2;
 RCC_OscInitStruct.PLL.PLLN = 20;
 RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV7;
 RCC_OscInitStruct.PLL.PLLQ = RCC_PLLQ_DIV2;
 RCC_OscInitStruct.PLL.PLLR = RCC_PLLR_DIV2;
 if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
 {
 Error_Handler();
 }

 RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSClk
 |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
 RCC_ClkInitStruct.SYSClkSource = RCC_SYSClkSOURCE_PLLCLK;
 RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSClk_DIV1;
 RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
 RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

 if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_4) != HAL_OK)
 {
 Error_Handler();
 }
}

```

```
}

if (HAL_PWREx_ControlVoltageScaling(PWR_REGULATOR_VOLTAGE_SCALE1) != HAL_OK)
{
 Error_Handler();
}
}

static void MX_TIM2_Init(void)
{
 TIM_ClockConfigTypeDef sClockSourceConfig = {0};
 TIM_MasterConfigTypeDef sMasterConfig = {0};

 htim2.Instance = TIM2;
 htim2.Init.Prescaler = 39999;
 htim2.Init.CounterMode = TIM_COUNTERMODE_DOWN;
 htim2.Init.Period = 1999;
 htim2.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
 htim2.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_ENABLE;
 if (HAL_TIM_Base_Init(&htim2) != HAL_OK)
 {
 Error_Handler();
 }
 sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
 if (HAL_TIM_ConfigClockSource(&htim2, &sClockSourceConfig) != HAL_OK)
 {
 Error_Handler();
 }
 sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
 sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
 if (HAL_TIMEx_MasterConfigSynchronization(&htim2, &sMasterConfig) != HAL_OK)
 {
 Error_Handler();
 }
}

static void MX_GPIO_Init(void)
{
 GPIO_InitTypeDef GPIO_InitStruct = {0};

 /* GPIO Ports Clock Enable */
 __HAL_RCC_GPIOA_CLK_ENABLE();

 /*Configure GPIO pin Output Level */
 HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_RESET);
```

```

/*Configure GPIO pin : PA5 */
GPIO_InitStruct.Pin = GPIO_PIN_5;
GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
GPIO_InitStruct.Pull = GPIO_NOPULL;
GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);

}

void Error_Handler(void)
{
}

#ifdef USE_FULL_ASSERT

void assert_failed(uint8_t *file, uint32_t line)
{
}

#endif

/***** (C) COPYRIGHT STMicroelectronics *****/

```

*Figure 6.4: Program: TMRLED.*

### Modified program

In the program in Figure 6.4, the code for the interrupt service routine is inside the **HAL\_TIM\_PeriodElapsedCallback** function. It is also possible to remove this code from the **Callback** function and put it inside function **TIM2\_IRQHandler** inside file **stm32l4xx\_itc.c** as follows:

```

void TIM2_IRQHandler(void)
{
 HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_5);
 HAL_TIM_IRQHandler(&htim2);
}

```

## 6.5 Project 2: 4-Digit 7-Segment LED Up Counter with Timer Interrupts

### Description

One of the problems when using a multiplexed 7-segment LED displays is that the processor must refresh the digits continuously and because of this the processor cannot do other tasks. In multiplexed 7-segment LED applications, the display refreshing code is usually placed in a timer interrupt service routine so that the display is refreshed in the background independently of the main program. Therefore, in this project we shall be placing the display code inside a timer interrupt service routine so that it is refreshed in the background and we can do other useful tasks in the main program. Here, a variable will be incremented every second in the main program and the count will be displayed on the LEDs. i.e. the

display will repeat counting from 0 to 9999 with one second delay between each count. The display is refreshed every 4 ms in this project. The timer parameters are set as follows.  $P = 39999$ ;  $I = 0.004$  second:

$$A = \frac{80,000,000 \times 0.004}{(39999 + 1)} - 1 = 7$$

### The aim

The aim of this project is to show how timer interrupts can be used to refresh a multiplexed 7-segment LED display.

### Block diagram

The block diagram of the project is as shown in Figure 5.75.

### Circuit diagram

The circuit diagram of the project is as shown in Figure 5.76.

### Program listing

In this program we will be using timer **TIM2** which is a general-purpose 32-bit timer. The steps are given below.

- Start STM32CubeIDE as before.
- Select the board type as Nucleo-L476RG.
- Give the name **TMRCNT** to the program.
- Configure PC0 to PC10 as digital output.
- Configure the MCU clock as 80 MHz and make sure that the APB1 clock is set to 80 MHz.
- Click **Timers**, then click TIM2, and select the **Clock Source** as **Internal clock** (Figure 6.3).
- Under **Configuration**, click on **Prescaler (PSC – 16-bit value)** and make sure it is set to **Decimal** and change its value to 39999.
- Change the **Counter Mode** to **Down**.
- Click on **Counter Period (Auto reload register)** and make sure it is set to **Decimal**. Change its value to 7.
- Click **Internal Clock Division** and set it to 2. Notice that this is not clock division, but it relates to the filter at the input sampler.
- Click **auto-reload preload** and click to **Enable** it
- Figure 6.5 shows the settings.

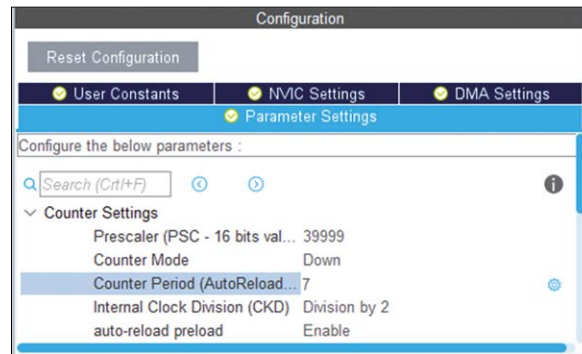


Figure 6.5: Timer settings.

- Click **System Core** followed by **NVIC**. Click to enable **TIM2 global interrupt**
- Click **File**, followed by **Save** and click **YES** to generate code.

You should now open the generated code. Insert the digit and the display bit patterns to the beginning of the program:

```
#define digit1 GPIO_PIN_7
#define digit2 GPIO_PIN_8
#define digit3 GPIO_PIN_9
#define digit4 GPIO_PIN_10
```

```
int LEDS[] = {0x3F, /* 0 */
 0x06, /* 1 */
 0x5B, /* 2 */
 0x4F, /* 3 */
 0x66, /* 4 */
 0x6D, /* 5 */
 0x7D, /* 6 */
 0x07, /* 7 */
 0x7F, /* 8 */
 0x6F, /* 9 */
 };
```

```
int LSD, MID1, MID2, MSD, m, n, flag = 0, Count = 0;
```

The interrupt service routine displays each digit for 4 ms. A variable called **flag** is used to determine which digit to enable. For example, digit1 is controlled as follows:

```
if(flag == 0)
{
 HAL_GPIO_WritePin(GPIOC, digit4, GPIO_PIN_RESET);
 GPIOC -> ODR = LEDS[MSD];
 HAL_GPIO_WritePin(GPIOC, digit1, GPIO_PIN_SET);
}
```



```
 flag++;
 }
}
```

Variable **flag** is incremented so that digit2 is enabled next time, and so on. Figure 6.6 shows the complete program listing (comments are removed).

```
/* USER CODE BEGIN Header */
/**

* @file : main.c
* @brief : Main program body

* @attention
*
* <h2><center>© Copyright (c) 2020 STMicroelectronics.
* All rights reserved.</center></h2>
*
* This software component is licensed by ST under BSD 3-Clause license,
* the «License»; You may not use this file except in compliance with the
* License. You may obtain a copy of the License at:
*
* opensource.org/licenses/BSD-3-Clause
*

*/
#include «main.h»

TIM_HandleTypeDef htim2;

#define digit1 GPIO_PIN_7
#define digit2 GPIO_PIN_8
#define digit3 GPIO_PIN_9
#define digit4 GPIO_PIN_10

int LEDS[] = {0x3F, /* 0 */
 0x06, /* 1 */
 0x5B, /* 2 */
 0x4F, /* 3 */
 0x66, /* 4 */
 0x6D, /* 5 */
 0x7D, /* 6 */
 0x07, /* 7 */
 0x7F, /* 8 */
 0x6F, /* 9 */
 };

int LSD, MID1, MID2, MSD, m, n, flag = 0, Count = 0;
```

```

void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_TIM2_Init(void);

//
// This is the timer interrupt service routine. The display is refreshed here
//
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
 MSD = Count / 1000; // Get MSD
 m = Count % 1000;
 MID2 = m / 100; // Get MID2
 n = m % 100;
 MID1 = n / 10; // Get MID1
 LSD = n % 10; // Get LSD

 if(flag == 0)
 {
 HAL_GPIO_WritePin(GPIOC, digit4, GPIO_PIN_RESET); // Disable digit4
 GPIOC -> ODR = LEDS[MSD]; // Output MSD
 HAL_GPIO_WritePin(GPIOC, digit1, GPIO_PIN_SET); // Enable digit1
 flag++;
 }

 else if(flag == 1)
 {
 HAL_GPIO_WritePin(GPIOC, digit1, GPIO_PIN_RESET); // Disable digit1
 GPIOC -> ODR = LEDS[MID2]; // Output MID2
 HAL_GPIO_WritePin(GPIOC, digit2, GPIO_PIN_SET); // Enable digit2
 flag++;
 }

 else if(flag == 2)
 {
 HAL_GPIO_WritePin(GPIOC, digit2, GPIO_PIN_RESET); // Disable digit2
 GPIOC -> ODR = LEDS[MID1]; // Output MID1
 HAL_GPIO_WritePin(GPIOC, digit3, GPIO_PIN_SET); // Enable digit3
 flag++;
 }

 else if(flag == 3)
 {
 HAL_GPIO_WritePin(GPIOC, digit3, GPIO_PIN_RESET); // Disable digit3
 GPIOC -> ODR = LEDS[LSD]; // Output LSD
 HAL_GPIO_WritePin(GPIOC, digit4, GPIO_PIN_SET); // Enable digit4
 }
}

```

```
 flag = 0;
 }
}

//
// Start of main program
//
int main(void)
{
 HAL_Init();

 SystemClock_Config();

 MX_GPIO_Init();
 MX_TIM2_Init();
 HAL_GPIO_WritePin(GPIOC, digit1, GPIO_PIN_RESET);
 HAL_GPIO_WritePin(GPIOC, digit2, GPIO_PIN_RESET);
 HAL_GPIO_WritePin(GPIOC, digit3, GPIO_PIN_RESET);
 HAL_GPIO_WritePin(GPIOC, digit4, GPIO_PIN_RESET);

 HAL_TIM_Base_Start_IT(&htim2);

 while (1)
 {
 Count++;
 if(Count > 9999)Count = 0;
 HAL_Delay(1000);
 }
}

void SystemClock_Config(void)
{
 RCC_OscInitTypeDef RCC_OscInitStruct = {0};
 RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

 RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
 RCC_OscInitStruct.HSISState = RCC_HSI_ON;
 RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
 RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
 RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
 RCC_OscInitStruct.PLL.PLLM = 2;
 RCC_OscInitStruct.PLL.PLLN = 20;
 RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV7;
 RCC_OscInitStruct.PLL.PLLQ = RCC_PLLQ_DIV2;
 RCC_OscInitStruct.PLL.PLLR = RCC_PLLR_DIV2;
```

```

if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
{
 Error_Handler();
}

RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
 |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_4) != HAL_OK)
{
 Error_Handler();
}

if (HAL_PWREx_ControlVoltageScaling(PWR_REGULATOR_VOLTAGE_SCALE1) != HAL_OK)
{
 Error_Handler();
}
}

static void MX_TIM2_Init(void)
{
 TIM_ClockConfigTypeDef sClockSourceConfig = {0};
 TIM_MasterConfigTypeDef sMasterConfig = {0};

 htim2.Instance = TIM2;
 htim2.Init.Prescaler = 39999;
 htim2.Init.CounterMode = TIM_COUNTERMODE_DOWN;
 htim2.Init.Period = 7;
 htim2.Init.ClockDivision = TIM_CLOCKDIVISION_DIV2;
 htim2.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_ENABLE;
 if (HAL_TIM_Base_Init(&htim2) != HAL_OK)
 {
 Error_Handler();
 }
 sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
 if (HAL_TIM_ConfigClockSource(&htim2, &sClockSourceConfig) != HAL_OK)
 {
 Error_Handler();
 }
 sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
 sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
 if (HAL_TIMEx_MasterConfigSynchronization(&htim2, &sMasterConfig) != HAL_OK)

```

```
{
 Error_Handler();
}

static void MX_GPIO_Init(void)
{
 GPIO_InitTypeDef GPIO_InitStruct = {0};

 /* GPIO Ports Clock Enable */
 __HAL_RCC_GPIOC_CLK_ENABLE();

 /*Configure GPIO pin Output Level */
 HAL_GPIO_WritePin(GPIOC, GPIO_PIN_0|GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3
 |GPIO_PIN_4|GPIO_PIN_5|GPIO_PIN_6|GPIO_PIN_7
 |GPIO_PIN_8|GPIO_PIN_9|GPIO_PIN_10, GPIO_PIN_RESET);

 /*Configure GPIO pins : PC0 PC1 PC2 PC3
 PC4 PC5 PC6 PC7
 PC8 PC9 PC10 */
 GPIO_InitStruct.Pin = GPIO_PIN_0|GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3
 |GPIO_PIN_4|GPIO_PIN_5|GPIO_PIN_6|GPIO_PIN_7
 |GPIO_PIN_8|GPIO_PIN_9|GPIO_PIN_10;
 GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
 GPIO_InitStruct.Pull = GPIO_NOPULL;
 GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
 HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);

}

void Error_Handler(void)
{
}

#ifdef USE_FULL_ASSERT

void assert_failed(uint8_t *file, uint32_t line)
{
}

#endif

/***** (C) COPYRIGHT STMicroelectronics *****END OF FILE*****/
```

*Figure 6.6: Program: TMRCNT.*

### Modified program

In the program in Figure 6.6, numbers are displayed with leading zeros. For example, number 1 is displayed as 0001, or number 25 is displayed as 0025. In many applications we may want to blank the leading zeroes. This can be done by enabling a digit depending on the value of the number to be displayed. For example, the MSD digit can only be enabled if the number is greater than 999, otherwise the digit is disabled which blanks the MSD part of the display:

```

 if(flag == 0)
 {
 if(Count > 999)
 {
 HAL_GPIO_WritePin(GPIOC, digit4, GPIO_PIN_RESET);
 GPIOC -> ODR = LEDS[MSD];
 HAL_GPIO_WritePin(GPIOC, digit1, GPIO_PIN_SET);
 }
 flag++;
 }

 else if(flag == 1)
 {
 if(Count > 99)
 {
 HAL_GPIO_WritePin(GPIOC, digit1, GPIO_PIN_RESET);
 GPIOC -> ODR = LEDS[MID2];
 HAL_GPIO_WritePin(GPIOC, digit2, GPIO_PIN_SET);
 }
 flag++;
 }

 else if(flag == 2)
 {
 if(Count > 9)
 {
 HAL_GPIO_WritePin(GPIOC, digit2, GPIO_PIN_RESET);
 GPIOC -> ODR = LEDS[MID1];
 HAL_GPIO_WritePin(GPIOC, digit3, GPIO_PIN_SET);
 }
 flag++;
 }

 else if(flag == 3)
 {
 HAL_GPIO_WritePin(GPIOC, digit3, GPIO_PIN_RESET);
 GPIOC -> ODR = LEDS[LSD];
 HAL_GPIO_WritePin(GPIOC, digit4, GPIO_PIN_SET);
 }

```

```
 flag = 0;
 }
```

## 6.6 Summary

In this Chapter we have learned how to use timers in our programs. An example is given to show how a 4-digit 7-segment display can be refreshed using timer interrupts.

In the next Chapter we will be learning how to use LCD displays in our programs.

## CHAPTER 7 • LCD Displays

### 7.1 Overview

In microcontroller systems the output of a measured variable is usually displayed using LEDs, 7-segment displays, or LCD type displays. LCDs have the advantages that they can be used to display alphanumeric or graphical data. Some LCDs have 40 or more character lengths with the capability to display several lines. Some other LCD displays can be used to display graphics images. Some modules offer colour displays while some others incorporate back lighting so that they can be viewed in dimly lit conditions.

There are basically two types of LCDs as far as the interface technique is concerned: parallel LCDs and serial LCDs. Parallel LCDs (e.g. Hitachi HD44780) are connected to a microcontroller using more than one data line and the data is transferred in parallel form. It is common to use either 4 or 8 data lines. Using a 4-wire connection saves I/O pins but it is slower since the data is transferred in two stages. Serial LCDs are connected to the microcontroller using only one data line and data is usually sent to the LCD using the standard RS-232 asynchronous data communication protocol. Serial LCDs are much easier to use but they cost more than the parallel ones.

The programming of a parallel LCD is usually a complex task and requires a good understanding of the internal operation of the LCD controllers, including the timing diagrams. Fortunately, most high-level languages provide special library commands for displaying data on alphanumeric as well as on graphical LCDs. All the user must do is connect the LCD to the microcontroller, define the LCD connection in software, and then send special commands to display data on the LCD.

In this Chapter we will be learning how to use LCDs in STM32 based projects.

### 7.2 Project 1: Using parallel LCDs – Displaying Text

#### Description

In this project we will be using a parallel LCD with our Nucleo-L476RG development board. The message **NUCLEO-L476 LCD** will be displayed on the LCD.

#### The HD44780 LCD module

HD44780 is one of the most popular alphanumeric LCD modules used in industry and also by hobbyists. This module is monochrome and comes in different sizes. Modules with 8, 16, 20, 24, 32, and 40 columns are available. Depending on the model chosen, the number of rows varies between 1, 2 or 4. The display provides a 14-pin (or 16-pin) connector to a microcontroller. Table 7.1 gives the pin configuration and pin functions of a 14-pin LCD module. Below is a summary of the pin functions:

| Pin no | Name | Function        |
|--------|------|-----------------|
| 1      | VSS  | Ground          |
| 2      | VDD  | + ve supply     |
| 3      | VEE  | Contrast        |
| 4      | RS   | Register select |
| 5      | R/W  | Read/write      |



|    |    |            |
|----|----|------------|
| 6  | E  | Enable     |
| 7  | D0 | Data bit 0 |
| 8  | D1 | Data bit 1 |
| 9  | D2 | Data bit 2 |
| 10 | D3 | Data bit 3 |
| 11 | D4 | Data bit 4 |
| 12 | D5 | Data bit 5 |
| 13 | D6 | Data bit 6 |
| 14 | D7 | Data bit 7 |

*Table 7.1: Pin configuration of HD44780 LCD module.*

- $V_{SS}$  is the 0 V supply or ground. The VDD pin should be connected to the positive supply. Although the manufacturers specify a 5 VDC supply, the modules will usually work with as low as 3 V or as high as 6 V.
- Pin 3 is named VEE and this is the contrast control pin. This pin is used to adjust the contrast of the display and it should be connected to a variable voltage supply. A potentiometer is normally connected between the power supply lines with its wiper arm connected to this pin allowing the contrast to be adjusted.
- Pin 4 is the Register Select (RS) and when this pin is LOW, data transferred to the display is treated as commands. When RS is HIGH, character data can be transferred to and from the module.
- Pin 5 is the Read/Write (R/W) line. This pin is pulled LOW in order to write commands or character data to the LCD module. When this pin is HIGH, character data or status information can be read from the module.
- Pin 6 is the Enable (E) pin which is used to initiate the transfer of commands or data between the module and the microcontroller. When writing to the display, data is transferred only on the HIGH to LOW transition of this line. When reading from the display, data becomes available after the LOW to HIGH transition of the enable pin and this data remains valid as long as the enable pin is at logic HIGH.
- Pins 7 to 14 are the eight data bus lines (D0 to D7). Data can be transferred between the microcontroller and the LCD module using either a single 8-bit byte, or as two 4-bit nibbles. In the latter case only the upper four data lines (D4 to D7) are used. 4-bit mode has the advantage that four less I/O lines are required to communicate with the LCD. In this book we shall be using alphanumeric based LCD only and look at the 4-bit interface only.

Figure 7.1 shows a typical HD44780 controller-based 2×16 (2 rows and 16 columns) character LCD. In this project we will display the text **NUCLEO-L476 LCD** at the first row of the LCD.



Figure 7.1: 2x16 character LCD.

### Connecting the LCD to the development board

The following pins are used in 4-bit mode:

D4:D7

E

R/S

In this project we shall be connecting the LCD to our development board as in the following table:

| LCD Pin | Microcontroller Pin |
|---------|---------------------|
| D4      | PA8                 |
| D5      | PA9                 |
| D6      | PA10                |
| D7      | PA11                |
| R/S     | PA6                 |
| E       | PA7                 |

### The aim

The aim of this project is to show how an LCD can be connected and used in Nucleo-L476 based projects.

### Block diagram

The block diagram of the project is shown in Figure 7.2.

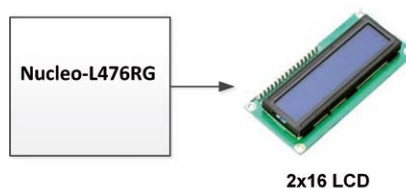


Figure 7.2: Block diagram of the project.

### Circuit diagram

The circuit diagram of the project is shown in Figure 7.3. The LCD is operated in 4-bit mode and the following connections are made between the LCD and the GPIO pins. The R/W pin of the LCD is not used and is connected to GND. The LCD is powered from the +5V supply, available on the morpho connectors and a 10-kΩ potentiometer is used at the  $V_{EE}$  pin to control the contrast of the LCD.

| LCD pin | Nucleo-L476 pin | CN10 pin |
|---------|-----------------|----------|
| R/S     | PA6             | 13       |
| E       | PA7             | 15       |
| D4      | PA8             | 23       |
| D5      | PA9             | 21       |
| D6      | PA10            | 33       |
| D7      | PA11            | 14       |

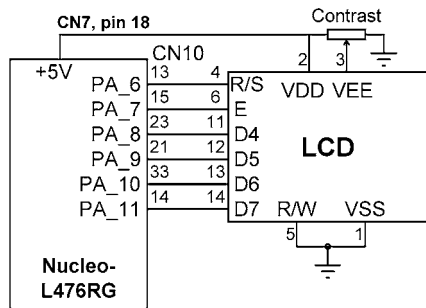


Figure 7.3: Circuit diagram of the project.

## Program listing

The steps are as follows.

- Start STM32CubeIDE.
- Create a new workspace.
- Click to start a new STM32 project.
- Search for STM32L476RG.
- Give the name LCDTEXT to the project.
- Configure PA6 to PA11 as digital output (Figure 7.4).

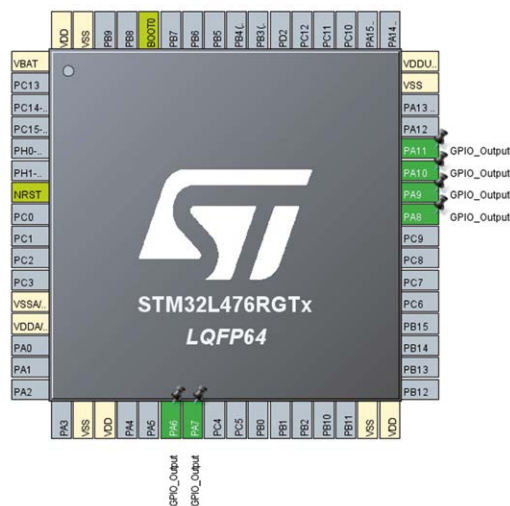


Figure 7.4: Configure GPIO.

- Configure the MCU clock for 80 MHz.
- Click **File**, followed by **Save** and click **YES** to generate the code.

In this project we will be using an LCD program created from the first principles by the author. This program initializes an HD44780 or a compatible LCD controller. Detailed operation of an LCD can be found in many references and tutorials on the Internet and it will not be repeated here. The LCD program is included at the beginning of the program.

GPIOA is used to control the LCD. The connections between the LCD and the Nucleo-L476RG board are defined at the beginning of the program and are as follows:

```
#define LCD_EN GPIO_PIN_7
#define LCD_RS GPIO_PIN_6
#define LCD_D4 GPIO_PIN_8
#define LCD_D5 GPIO_PIN_9
#define LCD_D6 GPIO_PIN_10
#define LCD_D7 GPIO_PIN_11
```

The following functions have been created to control the LCD:

**LCD\_STROBE:** This function sends a clock pulse to the LCD by rising and then lowering the enable pin of the LCD. This is required for sending a command or data to the LCD.

**lcd\_write\_cmd:** This function sends a command byte to the LCD. First the upper nibble is sent, followed by the lower nibble. Pin **RS** of the LCD is set LOW so that the data sent to the LCD is treated as a command. Notice that the byte to be sent is first copied to an integer variable called **d**. This variable is shifted left by 4 bits so that the upper nibble is at bit positions 8 to 11. The data is output to GPIOA with pin **RS** cleared to LOW. Function **LCD\_STROBE** is then called to clock the command to the LCD. This process is repeated for the lower nibble where the same variable is used, and it is shifted left by 8 bits so that the lower nibble is at bit positions 8 to 11.

**lcd\_write\_data:** This function is similar to **lcd\_write\_cmd** but here pin **RS** is set HIGH so that the data sent to the LCD is treated as data and not as command.

**lcd\_clear:** This function clears the LCD.

**lcd\_puts:** This function displays a string of characters on the LCD.

**lcd\_putchar:** This function displays a single character on the LCD.

**lcd\_goto:** This function positions the cursor at the given column and row coordinates. There are 2 rows, and each row has 16 columns. Coordinate (0, 0) is the home position, corresponding to the top left-hand pixel of the LCD.

**lcd\_init:** This function is used to initialize the LCD and it must be called before any other LCD function can be used. The LCD is configured to operate in 4-bit mode.

Figure 7.5 gives the complete program listing (program: **LCDTEXT**). The LCD code is at the beginning of the program and is enclosed by comments. It is possible to save the LCD code in files **LCD.c** and **LCD.h** and then include the source file in the project sources directory and the header file in the project include file so that there is no need to include the complete LCD code at the beginning of a program.

GPIOA has been configured as an output port. Inside the main program the LCD is initialized by calling function **lcd\_init**. The text message **NUCLEO-L476 LCD** is then displayed on the LCD by calling function **lcd\_puts**.

```

/* USER CODE BEGIN Header */
/**

 * @file : main.c
 * @brief : Main program body

 * @attention
 *
 * <h2><center>© Copyright (c) 2020 STMicroelectronics.
 * All rights reserved.</center></h2>
 *
 * This software component is licensed by ST under BSD 3-Clause license,
 * the «License»; You may not use this file except in compliance with the
 * License. You may obtain a copy of the License at:
 *
 * opensource.org/licenses/BSD-3-Clause
 *

 */
#include «main.h»
#include «LCD.h»

//
// ***** START OF LCD CODE *****
//
// In the following LCD code it is assumed that the LCD is connected to the
// Nucleo-L476RG board as follows in 4-bit mode:
//
// RS to PA6
// E to PA7
// D4 to PA8
// D5 to PA9
// D6 to PA10
// D7 to PA11
//
// LCD Header Code:
#define LCD_EN GPIO_PIN_7
#define LCD_RS GPIO_PIN_6

```

```

#define LCD_D4 GPIO_PIN_8
#define LCD_D5 GPIO_PIN_9
#define LCD_D6 GPIO_PIN_10
#define LCD_D7 GPIO_PIN_11

void LCD_STROBE(void);
void lcd_write_cmd(unsigned char);
void lcd_write_data(unsigned char);
void lcd_clear(void);
void lcd_puts(const char * s);
void lcd_putchar(char c);
void lcd_goto(int,int);
void lcd_init(void);

//
// Send clock pulse to the LCD
//
void LCD_STROBE()
{
 HAL_GPIO_WritePin(GPIOA,LCD_EN,GPIO_PIN_SET);
 HAL_Delay(0.1);
 HAL_GPIO_WritePin(GPIOA,LCD_EN,GPIO_PIN_RESET);
 HAL_Delay(0.1);
}

//
// Send a command to the LCD
//
void lcd_write_cmd(unsigned char c)
{
 unsigned int d;
 d = c;
 d = (d << 4) & 0x0F00; // Extract upper nibble
 GPIOA->ODR = d; // Output to GPIOA
 HAL_GPIO_WritePin(GPIOA,LCD_RS,GPIO_PIN_RESET); // Clear RS
 LCD_STROBE(); // Clock enable bit
 d = c;
 d = (d << 8) & 0x0F00; // Extract lower nibble
 GPIOA->ODR = d; // Output to GPIOA
 HAL_GPIO_WritePin(GPIOA,LCD_RS,GPIO_PIN_RESET); // Clear RS
 LCD_STROBE(); // Clock enable bit
 HAL_Delay(0.1);
}

//
// Send data to the LCD

```

```
//
void lcd_write_data(unsigned char c)
{
 unsigned int d;

 d = c;
 d = (d << 4) & 0x0F00; // Extract upper nibble
 GPIOA->ODR = d; // Output to GPIOA
 HAL_GPIO_WritePin(GPIOA,LCD_RS,GPIO_PIN_SET); // Set RS HIGH
 LCD_STROBE(); // Clock enable bit
 d = c;
 d = (d << 8) & 0x0F00; // Extract lower nibble
 GPIOA->ODR = d; // Output to GPIOA
 HAL_GPIO_WritePin(GPIOA,LCD_RS,GPIO_PIN_SET); // Set RS HIGH
 LCD_STROBE();
}

//
// Clear LCD
//
void lcd_clear(void)
{
 lcd_write_cmd(0x1);
 HAL_Delay(5);
}

//
// Display text message on LCD
//
void lcd_puts(const char * s)
{
 while(*s)
 lcd_write_data(*s++);
}

//
// Display single character on LCD
//
void lcd_putchar(char c)
{
 unsigned int d;
 d = c;
 d = (d << 4) & 0x0F00;
 GPIOA->ODR = d;
 HAL_GPIO_WritePin(GPIOA,LCD_RS,GPIO_PIN_SET);
 LCD_STROBE();
}
```

```

 d = c;
 d = (d << 8) & 0x0F00;
 GPIOA->ODR = d;
 HAL_GPIO_WritePin(GPIOA,LCD_RS,GPIO_PIN_SET);
 LCD_STROBE();
 }

 //
 // Position the cursor at column, row
 //
 void lcd_goto(int col, int row)
 {
 char address;
 if(row == 0)address = 0;
 if(row == 1)address = 0x40;
 address += col - 1;
 lcd_write_cmd(0x80 | address);
 }

 //
 // Initialize the LCD
 //
 void lcd_init(void)
 {
 GPIOA->ODR = 0;
 HAL_Delay(50);
 GPIOA->ODR = 0x0300;
 LCD_STROBE();
 HAL_Delay(30);
 LCD_STROBE();
 HAL_Delay(20);
 LCD_STROBE();
 HAL_Delay(20);
 GPIOA->ODR = 0x0200;
 LCD_STROBE();
 HAL_Delay(5);
 lcd_write_cmd(0x28);
 HAL_Delay(5);
 lcd_write_cmd(0x0F);
 HAL_Delay(5);
 lcd_write_cmd(0x01);
 HAL_Delay(5);
 lcd_write_cmd(0x06);
 HAL_Delay(5);
 }
 //

```



```
//***** END OF LCD CODE *****
//

void SystemClock_Config(void);
static void MX_GPIO_Init(void);

int main(void)
{
 HAL_Init();

 SystemClock_Config();

 MX_GPIO_Init();

 lcd_init();
 lcd_puts(«NUCLEO-L476 LCD»);

 while (1)
 {

 }
}

void SystemClock_Config(void)
{
 RCC_OscInitTypeDef RCC_OscInitStruct = {0};
 RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

 RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
 RCC_OscInitStruct.HSISState = RCC_HSI_ON;
 RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
 RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
 RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
 RCC_OscInitStruct.PLL.PLLM = 2;
 RCC_OscInitStruct.PLL.PLLN = 20;
 RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV7;
 RCC_OscInitStruct.PLL.PLLQ = RCC_PLLQ_DIV2;
 RCC_OscInitStruct.PLL.PLLR = RCC_PLLR_DIV2;
 if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
 {
 Error_Handler();
 }

 RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
 |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
 RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
}
```

```

RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_4) != HAL_OK)
{
 Error_Handler();
}

if (HAL_PWREx_ControlVoltageScaling(PWR_REGULATOR_VOLTAGE_SCALE1) != HAL_OK)
{
 Error_Handler();
}
}

static void MX_GPIO_Init(void)
{
 GPIO_InitTypeDef GPIO_InitStruct = {0};

 /* GPIO Ports Clock Enable */
 __HAL_RCC_GPIOA_CLK_ENABLE();

 /*Configure GPIO pin Output Level */
 HAL_GPIO_WritePin(GPIOA, GPIO_PIN_6|GPIO_PIN_7|GPIO_PIN_8|GPIO_PIN_9
 |GPIO_PIN_10|GPIO_PIN_11, GPIO_PIN_RESET);

 /*Configure GPIO pins : PA6 PA7 PA8 PA9
 PA10 PA11 */
 GPIO_InitStruct.Pin = GPIO_PIN_6|GPIO_PIN_7|GPIO_PIN_8|GPIO_PIN_9
 |GPIO_PIN_10|GPIO_PIN_11;
 GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
 GPIO_InitStruct.Pull = GPIO_NOPULL;
 GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
 HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
}

void Error_Handler(void)
{
}

#ifdef USE_FULL_ASSERT

void assert_failed(uint8_t *file, uint32_t line)
{
}

```

```
#endif
```

```
/***** (C) COPYRIGHT STMicroelectronics *****/
```

*Figure 7.5: Program: LCDTEXT.*

### LCD source file

As mentioned above, we can save the LCD source code in a file called **LCD.c** whose contents will be as shown in Figure 7.6.

```
#include «main.h»

//
// ***** START OF LCD CODE *****
//
// In the following LCD code it is assumed that the LCD is connected to the
// Nucleo-L476RG board as follows in 4-bit mode:
//
// RS to PA6
// E to PA7
// D4 to PA8
// D5 to PA9
// D6 to PA10
// D7 to PA11
//
// LCD Header Code:
#define LCD_EN GPIO_PIN_7
#define LCD_RS GPIO_PIN_6
#define LCD_D4 GPIO_PIN_8
#define LCD_D5 GPIO_PIN_9
#define LCD_D6 GPIO_PIN_10
#define LCD_D7 GPIO_PIN_11

//
// LCD C code:
void LCD_STROBE()
{
 HAL_GPIO_WritePin(GPIOA,LCD_EN,GPIO_PIN_SET);
 HAL_Delay(0.1);
 HAL_GPIO_WritePin(GPIOA,LCD_EN,GPIO_PIN_RESET);
 HAL_Delay(0.1);
}

//
// Send a command to the LCD
//
```

```

void lcd_write_cmd(unsigned char c)
{
 unsigned int d;
 d = c;
 d = (d << 4) & 0x0F00; // Extract upper nibble
 GPIOA->ODR = d; // Output to GPIOA
 HAL_GPIO_WritePin(GPIOA,LCD_RS,GPIO_PIN_RESET); // Clear RS
 LCD_STROBE(); // Clock enable bit
 d = c;
 d = (d << 8) & 0x0F00; // Extract lower nibble
 GPIOA->ODR = d; // Output to GPIOA
 HAL_GPIO_WritePin(GPIOA,LCD_RS,GPIO_PIN_RESET); // Clear RS
 LCD_STROBE(); // Clock enable bit
 HAL_Delay(0.1);
}

//
// Send data to the LCD
//
void lcd_write_data(unsigned char c)
{
 unsigned int d;

 d = c;
 d = (d << 4) & 0x0F00; // Extract upper nibble
 GPIOA->ODR = d; // Output to GPIOA
 HAL_GPIO_WritePin(GPIOA,LCD_RS,GPIO_PIN_SET); // Set RS HIGH
 LCD_STROBE(); // Clock enable bit
 d = c;
 d = (d << 8) & 0x0F00; // Extract lower nibble
 GPIOA->ODR = d; // Output to GPIOA
 HAL_GPIO_WritePin(GPIOA,LCD_RS,GPIO_PIN_SET); // Set RS HIGH
 LCD_STROBE();

 // Clock enable bit
}

//
// Clear LCD
//
void lcd_clear(void)
{
 lcd_write_cmd(0x1);
 HAL_Delay(5);
}

```

```
//
// Display text message on LCD
//
void lcd_puts(const char * s)
{
 while(*s)
 lcd_write_data(*s++);
}

//
// Display single character on LCD
//
void lcd_putchar(char c)
{
 unsigned int d;
 d = c;
 d = (d << 4) & 0x0F00;
 GPIOA->ODR = d;
 HAL_GPIO_WritePin(GPIOA,LCD_RS,GPIO_PIN_SET);
 LCD_STROBE();
 d = c;
 d = (d << 8) & 0x0F00;
 GPIOA->ODR = d;
 HAL_GPIO_WritePin(GPIOA,LCD_RS,GPIO_PIN_SET);
 LCD_STROBE();
}

//
// Position the cursor at column,row
//
void lcd_goto(int col, int row)
{
 char address;
 if(row == 0)address = 0;
 if(row == 1)address = 0x40;
 address += col - 1;
 lcd_write_cmd(0x80 | address);
}

//
// Initialize the LCD
//
void lcd_init(void)
{
 GPIOA->ODR = 0;
 HAL_Delay(50);
}
```

```

 GPIOA->ODR = 0x0300;
 LCD_STROBE();
 HAL_Delay(30);
 LCD_STROBE();
 HAL_Delay(20);
 LCD_STROBE();
 HAL_Delay(20);
 GPIOA->ODR = 0x0200;
 LCD_STROBE();
 HAL_Delay(5);
 lcd_write_cmd(0x28);
 HAL_Delay(5);
 lcd_write_cmd(0x0F);
 HAL_Delay(5);
 lcd_write_cmd(0x01);
 HAL_Delay(5);
 lcd_write_cmd(0x06);
 HAL_Delay(5);
}
//
//***** END OF LCD CODE *****
//

```

*Figure 7.6: Code for LCD.c.*

You should save **LCD.c** in the source directory of your project. e.g.:

C:\Users\username\STM32CubeIDE\worksspace\_1.3.18\LCDTEXT\Core\Src

### LCD header file

The contents of the LCD header file LCD.h are:

```

void LCD_STROBE(void);
void lcd_write_cmd(unsigned char);
void lcd_write_data(unsigned char);
void lcd_clear(void);
void lcd_puts(const char * s);
void lcd_putchar(char c);
void lcd_goto(int,int);
void lcd_init(void);

```

The header file should be saved in the include folder of the working directory:

C:\Users\username\STM32CubeIDE\worksspace\_1.3.18\LCDTEXT\Core\Inc

The username and workspace name will be different on your computer. Notice that the Project Explorer will display **LCD.c** when it is added to the sources directory (see Figure 7.7).

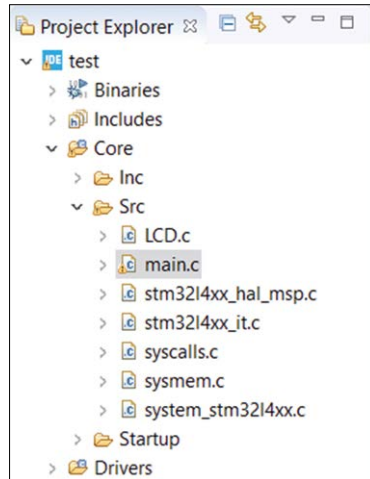


Figure 7.7: Project Explorer.

The following statement should be added to the beginning of the program:

**#include "LCD.h"**

Using **LCD.c**, the program in Figure 7.5 becomes much shorter as shown in Figure 7.8 (comments and the clock and error functions are not shown for clarity).

```
#include «main.h»
#include «LCD.h»

void SystemClock_Config(void);
static void MX_GPIO_Init(void);

int main(void)
{
 HAL_Init();
 SystemClock_Config();
 MX_GPIO_Init();

 lcd_init();
 lcd_puts(«NUCLEO-L476 LCD»);

 while (1)
 {
 }
}
```

```

static void MX_GPIO_Init(void)
{
 GPIO_InitTypeDef GPIO_InitStruct = {0};

 /* GPIO Ports Clock Enable */
 __HAL_RCC_GPIOA_CLK_ENABLE();

 /*Configure GPIO pin Output Level */
 HAL_GPIO_WritePin(GPIOA, GPIO_PIN_6|GPIO_PIN_7|GPIO_PIN_8|GPIO_PIN_9
 |GPIO_PIN_10|GPIO_PIN_11, GPIO_PIN_RESET);

 /*Configure GPIO pins : PA6 PA7 PA8 PA9
 PA10 PA11 */
 GPIO_InitStruct.Pin = GPIO_PIN_6|GPIO_PIN_7|GPIO_PIN_8|GPIO_PIN_9
 |GPIO_PIN_10|GPIO_PIN_11;
 GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
 GPIO_InitStruct.Pull = GPIO_NOPULL;
 GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
 HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);

}

void Error_Handler(void)
{
}

#ifdef USE_FULL_ASSERT

void assert_failed(uint8_t *file, uint32_t line)
{
}

#endif

/***** (C) COPYRIGHT STMicroelectronics *****END OF FILE*****/

```

*Figure 7.8: Using LCD.c.*

Compile the program in **Release** mode making sure that there are no errors. Drag and drop the binary file to device NUCLEO\_L476RG.

### 7.3 Project 2: Using LCDs – Simple Up Counter

#### Description

In this project an up counter is developed which counts up every second and the result is displayed on the LCD in the following format:



```
Count:
nn
```

Where the text **Count** is displayed at the beginning of row 0, and the number is displayed starting from row 1.

### The aim

The aim of this project is to show how both text and numbers can be displayed on the LCD.

### Block diagram

The block diagram of the project is as in Figure 7.2.

### Circuit diagram

The circuit diagram of the project is as in Figure 7.3.

### Program listing

Configure port pins PA6 to PA11 as digital outputs, and also configure the MCU clock to 80 MHz.

The program listing is shown in Figure 7.9 (program: **LDCCOUNT**). Make sure that you copy file **LCD.c** to the sources directory and **LCD.h** to include directory of your working project. A variable called **Count** is initialized to 0 at the beginning of the program. An array called **Txt** is created to store the count in string format. Count is incremented by one every second. It is then converted into a string array and stored in **Txt** using standard function **itoa**. Notice that library **stdlib.h** is included at the beginning of the program. String **Txt** is then displayed on the LCD in the required format.

```
/* USER CODE BEGIN Header */
/**

 * @file : main.c
 * @brief : Main program body

 * @attention
 *
 * <h2><center>© Copyright (c) 2020 STMicroelectronics.
 * All rights reserved.</center></h2>
 *
 * This software component is licensed by ST under BSD 3-Clause license,
 * the "License"; You may not use this file except in compliance with the
 * License. You may obtain a copy of the License at:
 *
 * opensource.org/licenses/BSD-3-Clause
 *

 */
#include "main.h"
#include "LCD.h"
```

```

#include "stdlib.h"

void SystemClock_Config(void);
static void MX_GPIO_Init(void);

int main(void)
{
 int Count = 0;
 char Txt[17];

 HAL_Init();

 SystemClock_Config();

 MX_GPIO_Init();
 lcd_init();
 lcd_puts("Count:"); // Heading

 while (1)
 {
 lcd_goto(0, 1); // Column 0, row 1
 itoa(Count, Txt, 10); // Convert to string
 lcd_puts(Txt); // Display Count
 Count++; // Increment Count
 HAL_Delay(1000); // Wait 1 second
 }
}

void SystemClock_Config(void)
{
 RCC_OscInitTypeDef RCC_OscInitStruct = {0};
 RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

 RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
 RCC_OscInitStruct.HSISState = RCC_HSI_ON;
 RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
 RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
 RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
 RCC_OscInitStruct.PLL.PLLM = 2;
 RCC_OscInitStruct.PLL.PLLN = 20;
 RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV7;
 RCC_OscInitStruct.PLL.PLLQ = RCC_PLLQ_DIV2;
 RCC_OscInitStruct.PLL.PLLR = RCC_PLLR_DIV2;
 if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
 {
 Error_Handler();
 }
}

```

```
}

RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
 |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_4) != HAL_OK)
{
 Error_Handler();
}

if (HAL_PWREx_ControlVoltageScaling(PWR_REGULATOR_VOLTAGE_SCALE1) != HAL_OK)
{
 Error_Handler();
}
}

static void MX_GPIO_Init(void)
{
 GPIO_InitTypeDef GPIO_InitStruct = {0};

 /* GPIO Ports Clock Enable */
 __HAL_RCC_GPIOA_CLK_ENABLE();

 /*Configure GPIO pin Output Level */
 HAL_GPIO_WritePin(GPIOA, GPIO_PIN_6|GPIO_PIN_7|GPIO_PIN_8|GPIO_PIN_9
 |GPIO_PIN_10|GPIO_PIN_11, GPIO_PIN_RESET);

 /*Configure GPIO pins : PA6 PA7 PA8 PA9
 PA10 PA11 */
 GPIO_InitStruct.Pin = GPIO_PIN_6|GPIO_PIN_7|GPIO_PIN_8|GPIO_PIN_9
 |GPIO_PIN_10|GPIO_PIN_11;
 GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
 GPIO_InitStruct.Pull = GPIO_NOPULL;
 GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
 HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
}

void Error_Handler(void)
{

```

```
}

#ifdef USE_FULL_ASSERT

void assert_failed(uint8_t *file, uint32_t line)
{
}
#endif

/***** (C) COPYRIGHT STMicroelectronics *****/
```

*Figure 7.9: Program: LCDCOUNT.*

## 7.4 Summary

In this Chapter we have learned how to use LCDs in projects. Next up in Chapter 7 we will start using various sensors in our projects.

## CHAPTER 8 • Using the Analogue to Digital Converters

### 8.1 Overview

In the last Chapter we have learned how to use LCDs in our projects. Currently most sensors produce analogue signals. For example, LM35 is a highly popular temperature sensor chip which produces analogue output voltage which is directly proportional to the measured temperature. Such signals can be read by the MCU by using analogue-to-digital converters (ADC). All STM32 microcontrollers offer several ADC channels. The analogue input voltage is compared with a fixed reference voltage. This reference voltage can be external or internal. In most applications it may be easier to use the internal reference voltage. Most STM32 microcontrollers provide 12-bit ADC. with +3.3 V reference voltage and 12-bit ADC, one LSB corresponds to:

$$3300 \text{ mV} / 4096 = 0.805 \text{ mV.}$$

In this Chapter we will be using the ADC in some projects.

### 8.2 The STM32 ADC conversion modes

Several conversion modes are provided by the STM32 ADC. A brief introduction to these modes is given in this section.

**Single conversion mode:** this is the simplest conversion mode. In this mode, the ADC performs the single conversion of a single channel and stops when conversion is finished. For example, this mode can be used to measure the temperature of a sensor. At the end of the conversion the EOC end-of-conversion flag is set.

**Single continuous conversion mode:** in this mode a single channel is converted continuously and indefinitely. Here the ADC works in the background, converting the input data continuously without any intervention from the CPU. At the end of the conversion the EOC end-of-conversion flag is set.

**Scan multi-channel conversion mode:** this mode is used to convert successively a number of channels. We can configure up to 16 channels with different sampling times and different orders.

**Scan multi-channel continuous conversion mode:** in this mode a number of channels are scanned continuously

**Discontinuous conversion mode:** in this mode, a few pre-defined channels are used and only a single channel is converted at each trigger signal.

Channels can be configured for regular (normal) scan or injection scan modes of operation. In regular scan mode, the conversion is repeated for all channels one by one. If an injected conversion is started, the ADC stops converting the regular conversion and proceeds to convert the injected channels. i.e. injected conversion has higher priority. Regular data conversion is stored in a 16-bit register. Injected data conversion is stored in 4 16-bit registers.

A channel can be triggered externally or internally by software. At the end of converting the injected channels, the conversion of the regular channels resume.

Up to 16 external ADC inputs are available on the STM32L476RG processor. Additionally, 2 internal channels are provided for on-chip analogue temperature sensor and battery voltage sensor. The ADC resolution can be up to 12-bits at 2.4 Msamples/second. By using triple interleaved mode, this can be extended to 7.2 Msamples/second.

Three ADC converters ADC1, ADC2, and ADC3 are available on the chip. ADC1 is the master, while ADC2 and ADC3 are slaves. The ADCs can be operated in Dual mode, where in this mode, two ADCs are used: an ADC master and an ADC slave. The beginning of conversion is triggered by the ADC master to the ADC slave. The converted data of both master and slave can be read in parallel by reading the multi-mode data register.

The following GPIO pins are available for these channels:

| <b>Pin</b> | <b>Channel</b> | <b>ADC</b> |
|------------|----------------|------------|
| PC0        | IN1            | ADC 1,2,3  |
| PC1        | IN2            | ADC 1,2,3  |
| PC2        | IN3            | ADC 1,2,3  |
| PC3        | IN4            | ADC 1,2,3  |
| PA0        | IN5            | ADC 1,2    |
| PA1        | IN6            | ADC 1,2    |
| PA2        | IN7            | ADC 1,2    |
| PA3        | IN8            | ADC 1,2    |
| PA4        | IN9            | ADC 1,2    |
| PA5        | IN10           | ADC 1,2    |
| PA6        | IN11           | ADC 1,2    |
| PA7        | IN12           | ADC 1,2    |
| PC4        | IN13           | ADC 1,2    |
| PC5        | IN14           | ADC 1,2    |
| PB0        | IN15           | ADC 1,2    |
| PB1        | IN16           | ADC 1,2    |

The ADC needs a minimum of 3 clock cycles for the sampling and 12 clock cycles for the conversion. For example, 12 bits are converted in 12 clock cycles. The sampling time can be programmed individually for each input channel. Longer sample times ensure that signals having a higher impedance are correctly converted. The ADC clock is derived from the system clock, or from the PLLSAI1 or the PLLSAI2 output. It can reach 80 MHz and can be divided by the following pre-scalers values: 1, 2, 4, 6, 8, 10, 12, 16, 32, 64, 128 or 256, it is asynchronous to the AHB clock. Alternatively, the ADC clock can be derived from the AHB clock of the ADC bus interface, divided by a programmable factor (1, 2 or 4).

The sampling time is the ADC clock cycles for which the Sample & Hold capacitor at the input of the ADC is charged up to the channel input voltage. Assuming that we use the minimum sampling time, which is 3 clock cycles, the conversion requires additional 12 cycles (for 12-bit conversion). Thus, 15 clock cycles are required for a complete conversion assuming the minimum sampling time is used. The sampling time is programmable over a large range (from 3 cycles minimum to 480 cycles maximum). Although the minimum is

3 cycles, some input signals may require longer sampling times. The longer the sampling time, the slower the ADC conversion rate will be. An example is given below.

Assume that the ADC clock frequency is set to 80 MHz. Assuming the ADC is configured for the minimum sampling time of 3 cycles, the sampling time will be:

$$T_{\text{sample}} = 3 / 80 \text{ MHz} = 37.5 \text{ ns}$$

The conversion takes 12 clock cycles, therefore,

$$T_{\text{conv}} = 12 / 80 \text{ MHz} = 150 \text{ ns}$$

The ADC conversion rate is:  $37.5 + 150 = 187.5 \text{ ns}$  which is equivalent to 5.34 MHz.

The ADC can be used in polling mode, in interrupt mode, or in DMA mode. An overrun flag can be generated if data is not read before the next converted data is ready. Each ADC can generate 4 different interrupts: end of conversion, end of injected conversion, analogue watchdog, and data overrun.

The input channels are fixed and bound to specific MCU pins. However, these channels can be logically reordered by assigning them an index called rank which ranges from 1 to 16. Rank 1 is the first channel to be sampled, rank 2, the second one and so on.

The ADCs are active in processor Run and Sleep modes. They are not available in Stop mode, but their contents are not destroyed. In Standby mode the ADCs are powered-down and must be re-initialized when power is returned.

The ADC can be driven in 3 modes: polling, interrupt, and DMA. In polling mode, the program waits until the conversion is complete. In interrupt mode, an interrupt is generated as soon as the conversion is complete. In DMA mode, the converted data is stored in memory without the intervention of the CPU.

Further detailed information on the ADCs can be obtained from the following STMicroelectronics' datasheet:

*RM0351 Reference Manual: STM32L47xxx, STM32L48xxx, STM32L49xxx and STM32L4Axxx advanced Arm-based 32-bit MCUs*

### 8.3 Project 1: Analogue Voltmeter (polling ADC)

#### Description

This is a simple project that shows how a single ADC channel can be used in continuous mode. In this project the analogue input voltage is measured and displayed on an LCD in millivolts.

#### The aim

The aim of this project is to show how a single channel ADC can be used in a simple project.

#### Block diagram

Figure 8.1 shows the block diagram of the project.



Figure 8.1: Block diagram of the project.

### Circuit diagram

Figure 8.2 shows the circuit diagram of the project. The LCD is connected to the development board as in Figure 7.3. The voltage to be measured is applied to analogue input PC2 (channel IN3).

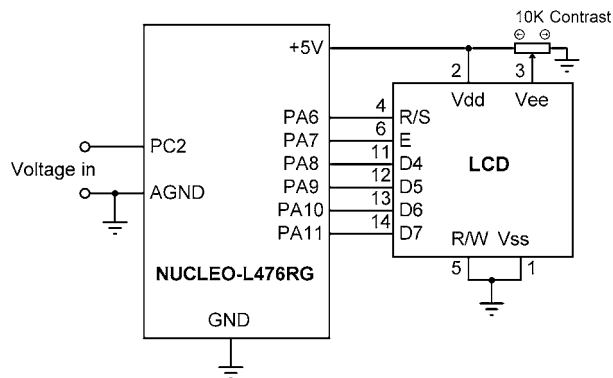


Figure 8.2: Circuit diagram of the project.

### Program listing

The steps are as follows.

- Start STM32CubeIDE as before.
- Create a new workspace.
- Select the board type as Nucleo-L476RG.
- Give the name **VOLTMETER** to the project.
- Configure PA6 to PA11 as digital output.
- Configure PC2 as ADC1\_IN3.
- Click the **Analog** tab and select **ADC1** and enable channel **IN3** as **Single-ended** to enable ADC on this channel (Pin PC2), see Figure 8.3.

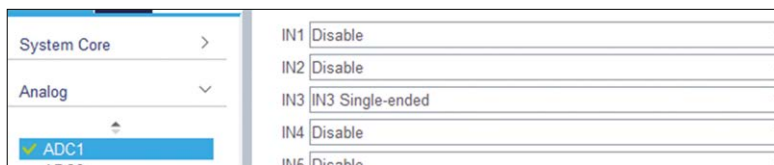


Figure 8.3: Select channel IN3 of ADC1.



- Under **Parameter Settings**, click on **Resolution** and select **ADC 12-bit resolution**. Also, enable the **continuous conversion mode**.
- Set **End of Conversion Selection** to **End of sequence of conversions**.
- Set **Overrun behaviour** to **Overrun data overwritten** (see Figure 8.4).

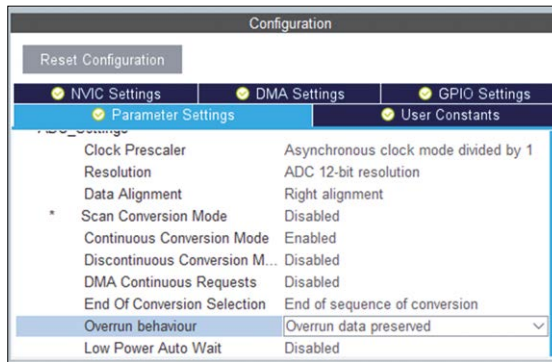


Figure 8.4: Parameter settings.

- Click **Clock Configuration** and make sure that the system clock and **ADC** clock are set as required
- Click **File**, followed by **Save** and click **YES** to generate code
- Click **Core**, followed by **Src** and double click **main.c** to open the main program

You should now copy file **LCD.c** to the **Src** directory and **LCD.h** to **Inc** directory of your STM32CubeIDE working directory.

The ADC initialization code is as follows.

```
static void MX_ADC1_Init(void)
{
 ADC_MultiModeTypeDef multimode = {0};
 ADC_ChannelConfTypeDef sConfig = {0};

 hadc1.Instance = ADC1;
 hadc1.Init.ClockPrescaler = ADC_CLOCK_ASYNC_DIV1;
 hadc1.Init.Resolution = ADC_RESOLUTION_12B;
 hadc1.Init.DataAlign = ADC_DATAALIGN_RIGHT;
 hadc1.Init.ScanConvMode = ADC_SCAN_DISABLE;
 hadc1.Init.EOCSelection = ADC_EOC_SEQ_CONV;
 hadc1.Init.LowPowerAutoWait = DISABLE;
 hadc1.Init.ContinuousConvMode = ENABLE;
 hadc1.Init.NbrOfConversion = 1;
 hadc1.Init.DiscontinuousConvMode = DISABLE;
 hadc1.Init.NbrOfDiscConversion = 1;
 hadc1.Init.ExternalTrigConv = ADC_SOFTWARE_START;
 hadc1.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_NONE;
```

```

hadc1.Init.DMAContinuousRequests = DISABLE;
hadc1.Init.Overrun = ADC_OVR_DATA_OVERWRITTEN;
hadc1.Init.OversamplingMode = DISABLE;
if (HAL_ADC_Init(&hadc1) != HAL_OK)
{
 Error_Handler();
}

multimode.Mode = ADC_MODE_INDEPENDENT;
if (HAL_ADCEx_MultiModeConfigChannel(&hadc1, &multimode) != HAL_OK)
{
 Error_Handler();
}

sConfig.Channel = ADC_CHANNEL_3;
sConfig.Rank = ADC_REGULAR_RANK_1;
sConfig.SamplingTime = ADC_SAMPLETIME_2CYCLES_5;
sConfig.SingleDiff = ADC_SINGLE_ENDED;
sConfig.OffsetNumber = ADC_OFFSET_NONE;
sConfig.Offset = 0;
if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
{
 Error_Handler();
}
}

```

Figure 8.5 shows the program listing. At the beginning of the program, a variable called **adcResult** is defined which will store the converted digital data. additionally, a floating point variable **mv** and a character array **buffer** are defined. Variable **mv** will store the converted data in millivolts, which will be converted into string and stored in **buffer**. The LCD is initialized by calling function **lcd\_init()**. The ADC conversion is then started and the program enters the **while** loop. Inside this loop, the ADC is polled for conversion, and the converted data is stored in **adcResult**. This variable is then multiplied by 3300.0 and divided by 4095.0 to convert the input data into millivolts (the reference voltage of the ADC is +3.3V and it is configured for 12 bits of operation). This result is then converted into string using the **sprint** function. Text **Voltmeter** is displayed at the top row of the LCD. The measured voltage in millivolts is displayed at the bottom row of the LCD. A typical display may look like:

```

Voltmeter
3246.85

```

The above process is repeated forever after one second of delay.

```
/* USER CODE BEGIN Header */
/**

 * @file : main.c
 * @brief : Main program body

 * @attention
 *
 * <h2><center>© Copyright (c) 2020 STMicroelectronics.
 * All rights reserved.</center></h2>
 *
 * This software component is licensed by ST under BSD 3-Clause license,
 * the «License»; You may not use this file except in compliance with the
 * License. You may obtain a copy of the License at:
 *
 * opensource.org/licenses/BSD-3-Clause
 *

 */
#include «main.h»
#include «LCD.h»
#include «stdio.h»

uint32_t adcResult = 0;

ADC_HandleTypeDef hadc1;

void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_ADC1_Init(void);

int main(void)
{
 float mv;
 char buff[10];

 HAL_Init();

 SystemClock_Config();

 MX_GPIO_Init();
 MX_ADC1_Init();
 lcd_init();

 HAL_ADC_Start(&hadc1);
 while (1)
 {
```

```

 HAL_ADC_PollForConversion(&hadc1, 100);
 adcResult = HAL_ADC_GetValue(&hadc1);
 mv = ((float)adcResult) * 3300.0 / 4095.0;
 lcd_clear();
 lcd_puts(«Voltmeter»);
 lcd_goto(0, 1);
 sprintf(buff, «%7.2f», mv);
 lcd_puts(buff);
 HAL_Delay(1000);
 }
}

void SystemClock_Config(void)
{
 RCC_OscInitTypeDef RCC_OscInitStruct = {0};
 RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};
 RCC_PeriphCLKInitTypeDef PeriphClkInit = {0};

 RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
 RCC_OscInitStruct.HSISState = RCC_HSI_ON;
 RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
 RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
 RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
 RCC_OscInitStruct.PLL.PLLM = 2;
 RCC_OscInitStruct.PLL.PLLN = 20;
 RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV7;
 RCC_OscInitStruct.PLL.PLLQ = RCC_PLLQ_DIV2;
 RCC_OscInitStruct.PLL.PLLR = RCC_PLLR_DIV2;
 if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
 {
 Error_Handler();
 }

 RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
 |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
 RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
 RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
 RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
 RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

 if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_4) != HAL_OK)
 {
 Error_Handler();
 }
 PeriphClkInit.PeriphClockSelection = RCC_PERIPHCLK_ADC;
 PeriphClkInit.AdcClockSelection = RCC_ADCCLKSOURCE_PLLSAI1;

```

```
PeriphClkInit.PLLSAI1.PLLSAI1Source = RCC_PLLSOURCE_HSI;
PeriphClkInit.PLLSAI1.PLLSAI1M = 2;
PeriphClkInit.PLLSAI1.PLLSAI1N = 8;
PeriphClkInit.PLLSAI1.PLLSAI1P = RCC_PLLP_DIV7;
PeriphClkInit.PLLSAI1.PLLSAI1Q = RCC_PLLQ_DIV2;
PeriphClkInit.PLLSAI1.PLLSAI1R = RCC_PLLR_DIV2;
PeriphClkInit.PLLSAI1.PLLSAI1ClockOut = RCC_PLLSAI1_ADC1CLK;
if (HAL_RCCEx_PeriphCLKConfig(&PeriphClkInit) != HAL_OK)
{
 Error_Handler();
}

if (HAL_PWREx_ControlVoltageScaling(PWR_REGULATOR_VOLTAGE_SCALE1) != HAL_OK)
{
 Error_Handler();
}
}

static void MX_ADC1_Init(void)
{
 ADC_MultiModeTypeDef multimode = {0};
 ADC_ChannelConfTypeDef sConfig = {0};

 hadc1.Instance = ADC1;
 hadc1.Init.ClockPrescaler = ADC_CLOCK_ASYNC_DIV1;
 hadc1.Init.Resolution = ADC_RESOLUTION_12B;
 hadc1.Init.DataAlign = ADC_DATAALIGN_RIGHT;
 hadc1.Init.ScanConvMode = ADC_SCAN_DISABLE;
 hadc1.Init.EOCSelection = ADC_EOC_SEQ_CONV;
 hadc1.Init.LowPowerAutoWait = DISABLE;
 hadc1.Init.ContinuousConvMode = ENABLE;
 hadc1.Init.NbrOfConversion = 1;
 hadc1.Init.DiscontinuousConvMode = DISABLE;
 hadc1.Init.NbrOfDiscConversion = 1;
 hadc1.Init.ExternalTrigConv = ADC_SOFTWARE_START;
 hadc1.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_NONE;
 hadc1.Init.DMAContinuousRequests = DISABLE;
 hadc1.Init.Overrun = ADC_OVR_DATA_OVERWRITTEN;
 hadc1.Init.OversamplingMode = DISABLE;
 if (HAL_ADC_Init(&hadc1) != HAL_OK)
 {
 Error_Handler();
 }

 multimode.Mode = ADC_MODE_INDEPENDENT;
 if (HAL_ADCEx_MultiModeConfigChannel(&hadc1, &multimode) != HAL_OK)
```

```

{
 Error_Handler();
}

sConfig.Channel = ADC_CHANNEL_3;
sConfig.Rank = ADC_REGULAR_RANK_1;
sConfig.SamplingTime = ADC_SAMPLETIME_2CYCLES_5;
sConfig.SingleDiff = ADC_SINGLE_ENDED;
sConfig.OffsetNumber = ADC_OFFSET_NONE;
sConfig.Offset = 0;
if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
{
 Error_Handler();
}
}

static void MX_GPIO_Init(void)
{
 GPIO_InitTypeDef GPIO_InitStruct = {0};

 /* GPIO Ports Clock Enable */
 __HAL_RCC_GPIOC_CLK_ENABLE();
 __HAL_RCC_GPIOA_CLK_ENABLE();

 /*Configure GPIO pin Output Level */
 HAL_GPIO_WritePin(GPIOA, GPIO_PIN_6|GPIO_PIN_7|GPIO_PIN_8|GPIO_PIN_9
 |GPIO_PIN_10|GPIO_PIN_11, GPIO_PIN_RESET);

 /*Configure GPIO pins : PA6 PA7 PA8 PA9
 PA10 PA11 */
 GPIO_InitStruct.Pin = GPIO_PIN_6|GPIO_PIN_7|GPIO_PIN_8|GPIO_PIN_9
 |GPIO_PIN_10|GPIO_PIN_11;
 GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
 GPIO_InitStruct.Pull = GPIO_NOPULL;
 GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
 HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
}

void Error_Handler(void)
{
}

#ifdef USE_FULL_ASSERT

void assert_failed(uint8_t *file, uint32_t line)

```

```

{
}
#endif

/***** (C) COPYRIGHT STMicroelectronics *****END OF FILE*****/

```

Figure 8.5: The VOLTMETER program.

Notice that by default the floating-point function sprint is disabled by the linker. This can be enabled as follows.

- Click **Project** followed by **Properties**.
- Click **C/C++** followed by **Build**, followed by **Settings** and then **Tool Settings**.
- Click **Miscellaneous**.
- Click **Add** (first icon) in **Other flags** and add the following statement:  
**-u\_printf\_float**
- Click **Apply and Close** (Figure 8.6).

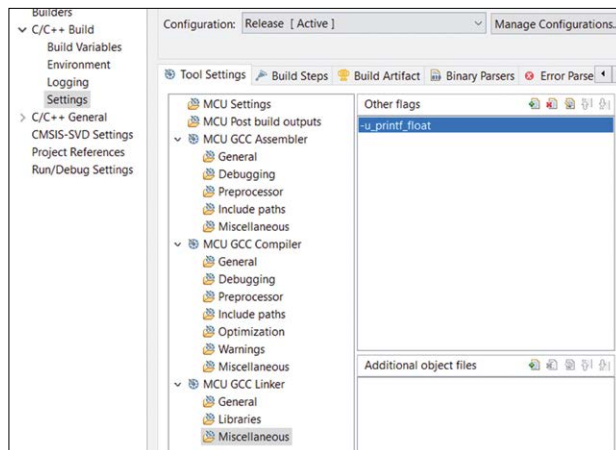


Figure 8.6: Add `-u_printf_float` to the linker.

Here is the code that starts the ADC reads the data, and displays it on the LCD every second.

```

HAL_ADC_Start(&hadc1);
while (1)
{
 HAL_ADC_PollForConversion(&hadc1, 100);
 adcResult = HAL_ADC_GetValue(&hadc1);
 mv = ((float)adcResult) * 3300.0 / 4095.0;
 lcd_clear();
 lcd_puts("Voltmeter");
 lcd_goto(0, 1);
}

```

```
 sprintf(buff, "%7.2f", mv);
 lcd_puts(buff);
 HAL_Delay(1000);
}
```

The following HAL functions are used in this program:

**HAL\_Init():** this function initializes the ADC.

**HAL\_ADC\_Start(&hadc1):** this function starts the ADC conversion.

**HAL\_ADC\_PollForConversion(&hadc1, 100):** this function waits for the ADC conversion to complete. 100 in this example is the timeout in milliseconds.

**adcResult = HAL\_ADC\_GetValue(&hadc1):** this function reads the converted ADC data into variable adcResult.

**HAL\_Delay(1000):** this function creates one second delay.

Some other useful HAL ADC functions are:

**HAL\_ADC\_Stop(handle):** this function stops the ADC conversion

**HAL\_ADC\_Start\_IT(handle):** this function starts ADC conversion of regular group with interruption

**HAL\_ADC\_Stop\_IT(handle):** this function stops ADC conversion when interruptions are enabled

**HAL\_ADC\_Start\_DMA(handle, data, length):** this function enabled ADC, starts conversion of regular group and transfers result through DMA

**HAL\_ADC\_Stop\_DMA(handle):** this function stops ADC conversion of regular groups and disables ADC DMA transfers

**HAL\_ADC\_IRQHandler(handle):** this function handles ADC interrupt requests

**HAL\_ADC\_ConvCpltCallback(handle):** this is the conversion complete callback in non-blocking mode

## 8.4 Project 2: ADC with Multiple Inputs (polling ADC)

### Description

In this project the analogue voltages at two inputs are measured and displayed on two rows of the LCD.



## The aim

The aim of this project is to show how multiple analogue inputs can be used in a project.

## Circuit diagram

Figure 8.7 shows the circuit diagram of the project. Analogue inputs PC2 (channel IN3) and PC3 (channel IN4) are used in this project.

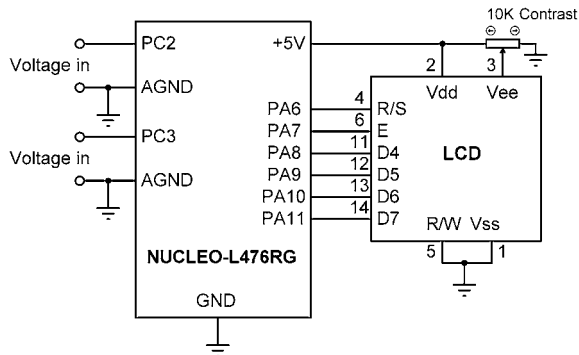


Figure 8.7: Circuit diagram of the project.

## Program listing

The steps are as follows:

- Start STM32CubeIDE as before.
- Create a new workspace.
- Select the board type as Nucleo-L476RG.
- Give the name **MULTIADC** to the project.
- Configure PA6 to PA11 as digital outputs for the LCD.
- Configure PC2 as ADC1\_IN3.
- Configure PC3 as ADC1\_IN4 (Figure 8.8).

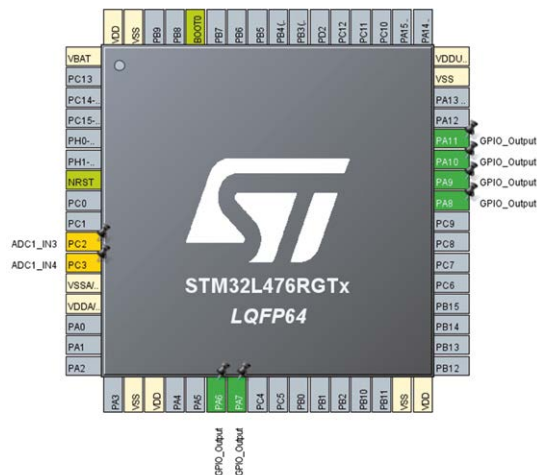


Figure 8.8: Configuring the GPIO.

- Click the **Analog** tab and select **ADC1** and enable channel **IN3** as **Single-ended** to enable ADC on this channel (Pin PC2). Similarly, enable **IN4** as **Single-ended** to enable ADC on this channel (PC3).
- Under **Parameter Settings**, click on **Resolution** and select **ADC 12-bit resolution**. Enable **Scan Conversion Mode**. Also, enable the **Continuous Conversion Mode**
- Set **End of Conversion Selection** to **End of sequence of conversions**.
- Set **Overrun behaviour** to **Overrun data overwritten**.
- Set **Number of Conversions** to **2**.
- Set **Rank 1 Channel** to **Channel 3**. Also, set **Rank 2 Channel** to **Channel 4** (Figure 8.9).

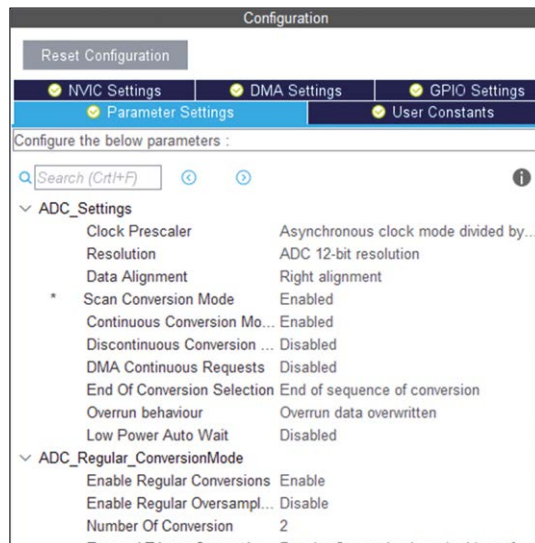


Figure 8.9: Configuration.

- Notice that in this project we are using only ADC1 with two inputs.
- Click **Clock Configuration** and make sure that the system clock and **ADC** clock are set as required.
- Click **File**, followed by **Save** and click **YES** to generate code.
- Click **Core**, followed by **Src** and double click **main.c** to open the main program.
- Copy file **LCD.c** to the sources directory and **LCD.h** to include directory of the STM32CubeIDE working directory.

Enter the following statements to store the converted data:

```
uint32_t adcResult1, adcResult2;
```

Enter statement **lcd\_init()** before the program loop to initialize the LCD. The ADC initialization routine contains the following data:

```

static void MX_ADC1_Init(void)
{
 ADC_MultiModeTypeDef multimode = {0};
 ADC_ChannelConfTypeDef sConfig = {0};

 hadc1.Instance = ADC1;
 hadc1.Init.ClockPrescaler = ADC_CLOCK_ASYNC_DIV1;
 hadc1.Init.Resolution = ADC_RESOLUTION_12B;
 hadc1.Init.DataAlign = ADC_DATAALIGN_RIGHT;
 hadc1.Init.ScanConvMode = ADC_SCAN_ENABLE;
 hadc1.Init.EOCSelection = ADC_EOC_SEQ_CONV;
 hadc1.Init.LowPowerAutoWait = DISABLE;
 hadc1.Init.ContinuousConvMode = ENABLE;
 hadc1.Init.NbrOfConversion = 2;
 hadc1.Init.DiscontinuousConvMode = DISABLE;
 hadc1.Init.NbrOfDiscConversion = 1;
 hadc1.Init.ExternalTrigConv = ADC_SOFTWARE_START;
 hadc1.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_NONE;
 hadc1.Init.DMAContinuousRequests = DISABLE;
 hadc1.Init.Overrun = ADC_OVR_DATA_OVERWRITTEN;
 hadc1.Init.OversamplingMode = DISABLE;

```

The two ADC channels (Channel 3 and Channel 4) are initialized as follows:

```

sConfig.Channel = ADC_CHANNEL_3;
sConfig.Rank = ADC_REGULAR_RANK_1;
sConfig.SamplingTime = ADC_SAMPLETIME_2CYCLES_5;
sConfig.SingleDiff = ADC_SINGLE_ENDED;
sConfig.OffsetNumber = ADC_OFFSET_NONE;
sConfig.Offset = 0;
if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
{
 Error_Handler();
}

sConfig.Channel = ADC_CHANNEL_4;
sConfig.Rank = ADC_REGULAR_RANK_2;
if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
{
 Error_Handler();
}

```

The program loop starts the ADC converter, reads the ADC data for each input, then stops the ADC. The received digital data is converted into millivolts and displayed on the top and bottom rows of the LCD. The statements inside the program loop are:

```

while (1)
{
 HAL_ADC_Start(&hadc1); // Start ADC
 HAL_ADC_PollForConversion(&hadc1, 100); // Poll ADC
 adcResult1 = HAL_ADC_GetValue(&hadc1); // Get the result

 HAL_ADC_PollForConversion(&hadc1, 100); // Poll the ADC
 adcResult2 = HAL_ADC_GetValue(&hadc1); // Get next result
 HAL_ADC_Stop(&hadc1); // Stop ADC

 mv = ((float)adcResult1) * 3300.0 / 4095.0; // Convert to mV
 lcd_clear(); // Clear LCD
 lcd_goto(0, 0); // To top row
 sprintf(buff, "%7.2f", mv); // Convert to string
 lcd_puts(buff); // Display

 mv = ((float)adcResult2) * 3300.0 / 4095.0; // Convert to mv
 lcd_goto(0, 1); // To bottom row
 sprintf(buff, "%7.2f", mv); // Convert to string
 lcd_puts(buff); // Display
 HAL_Delay(1000); // Wait 1 second
}

```

Compile the program in Release mode and make sure there are no errors. Drag and drop the binary file **MULTIADC.bin** to drive NUCLEO\_L476RG.

Figure 8.10 shows the complete program listing (comments are removed for clarity).

```

/* USER CODE BEGIN Header */
/**
 * *****
 * @file : main.c
 * @brief : Main program body
 * *****
 * @attention
 *
 * <h2><center>© Copyright (c) 2020 STMicroelectronics.
 * All rights reserved.</center></h2>
 *
 * This software component is licensed by ST under BSD 3-Clause license,
 * the «License»; You may not use this file except in compliance with the
 * License. You may obtain a copy of the License at:
 *
 * opensource.org/licenses/BSD-3-Clause
 *
 * *****
 */
#include «main.h»

```

```
#include «LCD.h»
#include «stdio.h»

uint32_t adcResult1, adcResult2;

ADC_HandleTypeDef hadc1;

void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_ADC1_Init(void);

int main(void)
{
 float mv;
 char buff[10];

 HAL_Init();

 SystemClock_Config();

 MX_GPIO_Init();
 MX_ADC1_Init();
 lcd_init();

 while (1)
 {
 HAL_ADC_Start(&hadc1); // Start ADC
 HAL_ADC_PollForConversion(&hadc1, 100); // Poll ADC
 adcResult1 = HAL_ADC_GetValue(&hadc1); // Get the result

 HAL_ADC_PollForConversion(&hadc1, 100); // Poll the ADC
 adcResult2 = HAL_ADC_GetValue(&hadc1); // Get next result
 HAL_ADC_Stop(&hadc1); // Stop ADC

 mv = ((float)adcResult1) * 3300.0 / 4095.0; // Convert to mV
 lcd_clear(); // Clear LCD
 lcd_goto(0, 0); // To top row
 sprintf(buff, «%7.2f», mv); // Convert to string
 lcd_puts(buff); // Display

 mv = ((float)adcResult2) * 3300.0 / 4095.0; // Convert to mv
 lcd_goto(0, 1); // To bottom row
 sprintf(buff, «%7.2f», mv); // Convert to string
 lcd_puts(buff); // Display
 HAL_Delay(1000); // Wait 1 second
 }
}
```

```

}

void SystemClock_Config(void)
{
 RCC_OscInitTypeDef RCC_OscInitStruct = {0};
 RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};
 RCC_PeriphCLKInitTypeDef PeriphClkInit = {0};

 RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
 RCC_OscInitStruct.HSIState = RCC_HSI_ON;
 RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
 RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
 RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
 RCC_OscInitStruct.PLL.PLLM = 2;
 RCC_OscInitStruct.PLL.PLLN = 20;
 RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV7;
 RCC_OscInitStruct.PLL.PLLQ = RCC_PLLQ_DIV2;
 RCC_OscInitStruct.PLL.PLLR = RCC_PLLR_DIV2;
 if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
 {
 Error_Handler();
 }

 RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
 |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
 RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
 RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
 RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
 RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

 if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_4) != HAL_OK)
 {
 Error_Handler();
 }
 PeriphClkInit.PeriphClockSelection = RCC_PERIPHCLK_ADC;
 PeriphClkInit.AdcClockSelection = RCC_ADCCLKSOURCE_PLLSAI1;
 PeriphClkInit.PLLSAI1.PLLSAI1Source = RCC_PLLSOURCE_HSI;
 PeriphClkInit.PLLSAI1.PLLSAI1M = 2;
 PeriphClkInit.PLLSAI1.PLLSAI1N = 8;
 PeriphClkInit.PLLSAI1.PLLSAI1P = RCC_PLLP_DIV7;
 PeriphClkInit.PLLSAI1.PLLSAI1Q = RCC_PLLQ_DIV2;
 PeriphClkInit.PLLSAI1.PLLSAI1R = RCC_PLLR_DIV2;
 PeriphClkInit.PLLSAI1.PLLSAI1ClockOut = RCC_PLLSAI1_ADC1CLK;
 if (HAL_RCCEx_PeriphCLKConfig(&PeriphClkInit) != HAL_OK)
 {
 Error_Handler();
 }
}

```

```
}

if (HAL_PWREx_ControlVoltageScaling(PWR_REGULATOR_VOLTAGE_SCALE1) != HAL_OK)
{
 Error_Handler();
}
}

static void MX_ADC1_Init(void)
{
 ADC_MultiModeTypeDef multimode = {0};
 ADC_ChannelConfTypeDef sConfig = {0};

 hadc1.Instance = ADC1;
 hadc1.Init.ClockPrescaler = ADC_CLOCK_ASYNC_DIV1;
 hadc1.Init.Resolution = ADC_RESOLUTION_12B;
 hadc1.Init.DataAlign = ADC_DATAALIGN_RIGHT;
 hadc1.Init.ScanConvMode = ADC_SCAN_ENABLE;
 hadc1.Init.EOCSelection = ADC_EOC_SEQ_CONV;
 hadc1.Init.LowPowerAutoWait = DISABLE;
 hadc1.Init.ContinuousConvMode = ENABLE;
 hadc1.Init.NbrOfConversion = 2;
 hadc1.Init.DiscontinuousConvMode = DISABLE;
 hadc1.Init.NbrOfDiscConversion = 1;
 hadc1.Init.ExternalTrigConv = ADC_SOFTWARE_START;
 hadc1.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_NONE;
 hadc1.Init.DMAContinuousRequests = DISABLE;
 hadc1.Init.Overrun = ADC_OVR_DATA_OVERWRITTEN;
 hadc1.Init.OversamplingMode = DISABLE;
 if (HAL_ADC_Init(&hadc1) != HAL_OK)
 {
 Error_Handler();
 }

 multimode.Mode = ADC_MODE_INDEPENDENT;
 if (HAL_ADCEx_MultiModeConfigChannel(&hadc1, &multimode) != HAL_OK)
 {
 Error_Handler();
 }

 sConfig.Channel = ADC_CHANNEL_3;
 sConfig.Rank = ADC_REGULAR_RANK_1;
 sConfig.SamplingTime = ADC_SAMPLETIME_2CYCLES_5;
 sConfig.SingleDiff = ADC_SINGLE_ENDED;
 sConfig.OffsetNumber = ADC_OFFSET_NONE;
 sConfig.Offset = 0;
```

```
if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
{
 Error_Handler();
}

sConfig.Channel = ADC_CHANNEL_4;
sConfig.Rank = ADC_REGULAR_RANK_2;
if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
{
 Error_Handler();
}
}

static void MX_GPIO_Init(void)
{
 GPIO_InitTypeDef GPIO_InitStruct = {0};

 /* GPIO Ports Clock Enable */
 __HAL_RCC_GPIOC_CLK_ENABLE();
 __HAL_RCC_GPIOA_CLK_ENABLE();

 /*Configure GPIO pin Output Level */
 HAL_GPIO_WritePin(GPIOA, GPIO_PIN_6|GPIO_PIN_7|GPIO_PIN_8|GPIO_PIN_9
 |GPIO_PIN_10|GPIO_PIN_11, GPIO_PIN_RESET);

 /*Configure GPIO pins : PA6 PA7 PA8 PA9
 PA10 PA11 */
 GPIO_InitStruct.Pin = GPIO_PIN_6|GPIO_PIN_7|GPIO_PIN_8|GPIO_PIN_9
 |GPIO_PIN_10|GPIO_PIN_11;
 GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
 GPIO_InitStruct.Pull = GPIO_NOPULL;
 GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
 HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
}

void Error_Handler(void)
{
}

#ifdef USE_FULL_ASSERT

void assert_failed(uint8_t *file, uint32_t line)
{
}

#endif
```



```
/***** (C) COPYRIGHT STMicroelectronics *****/
```

*Figure 8.10: The MULTIADC program.*

### 8.5 Project 3: Single-input ADC with Conversion Interrupt

#### Description

In the previous projects we have been using the ADC in polling mode. In this project the external analogue voltage to be measured is connected to analogue input PC2 as in Project 1. The program jumps to an interrupt service routine whenever the ADC completes a conversion. The measured voltage is displayed on the LCD. Interrupts based programs provide faster response.

#### The aim

The aim of this project is to show how the ADC completion interrupt can be used in a program.

#### Block Diagram

The block diagram of the project is same as in Figure 8.1.

#### Circuit Diagram

The circuit diagram of the project is same as in Figure 8.2 where the analogue voltage to be measured is applied to PC2.

#### Program listing

The steps are as follows.

- Start STM32CubeIDE as before.
- Create a new workspace.
- Select the board type as Nucleo-L476RG.
- Give the name **ADCINT** to the project.
- Configure PA6 to PA11 as digital outputs for the LCD.
- Configure the clock for 80 MHz.
- Configure PC2 as ADC1\_IN3.
- Click the **Analog** tab and select **ADC1** and enable channel **IN3** as **Single-ended** to enable ADC on this channel.
- Under **Parameter Settings**, click on **Resolution** and select **ADC 12-bit resolution**. Also, disable the **Continuous Conversion Mode**.
- Set **End of Conversion Selection** to **End of single of conversion**.
- Set **Overrun behaviour** to **Overrun data overwritten**.
- Click tab **System Core** and then click **NVIC**.
- Click to **enable** ADC1 interrupt.
- Click **File**, followed by **Save** and click **YES** to generate code.
- Click **Core**, followed by **Src** and double click **main.c** to open the main program.
- Copy file **LCD.c** to the **Src** directory and **LCD.h** to **Inc** directory of the STM32CubeIDE working directory.

Include **LCD.h** and **stdio.h** at the beginning of the program. Variable **convend** is set to 1 when an ADC conversion is complete. This variable is used in the main program to read and display the converted value on the LCD. The ADC is started with statement:

```
HAL_ADC_Start_IT(&hadc1)
```

The program loop reads the converted value, re-starts the ADC, converts the reading into millivolts, displays a heading at the top row, converts the reading into string, and then displays on the LCD. This process is repeated after one second delay.

Figure 8.11 shows the complete program listing (comments are removed).

```
/* USER CODE BEGIN Header */
/**
 * *****
 * @file : main.c
 * @brief : Main program body
 * *****
 * @attention
 *
 * <h2><center>© Copyright (c) 2020 STMicroelectronics.
 * All rights reserved.</center></h2>
 *
 * This software component is licensed by ST under BSD 3-Clause license,
 * the «License»; You may not use this file except in compliance with the
 * License. You may obtain a copy of the License at:
 *
 * opensource.org/licenses/BSD-3-Clause
 *
 * *****
 */
#include «main.h»
#include «LCD.h»
#include «stdio.h»

int convend = 0;
ADC_HandleTypeDef hadc1;

void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_ADC1_Init(void);

//
// This is the ADC conversion completion interrupt
//
void HAL_ADCx_ConvCpltCallback(ADC_HandleTypeDef* hadc1)
{
```

```
 convend = 1; // ADC completed
 }

//
// Start of main program
//
int main(void)
{
 uint32_t adcResult;
 char buff[10];
 float mv;

 HAL_Init();

 MX_GPIO_Init();
 MX_ADC1_Init();
 lcd_init();

 HAL_ADC_Start_IT(&hadc1);

 while (1)
 {
 if(convend == 1) // If ADC complete
 {
 convend=0; // Reset complete flag
 adcResult = HAL_ADC_GetValue(&hadc1); // Get result
 HAL_ADC_Start_IT(&hadc1); // Re-start ADC
 mv = ((float)adcResult) * 3300.0 / 4095.0; // Convert to mv
 lcd_clear(); // Clear lcd
 lcd_puts("Voltmeter"); // Heading
 lcd_goto(0, 1); // Bottom row
 sprintf(buff, "%7.2f", mv); // Convert to string
 lcd_puts(buff); // DIsplay
 HAL_Delay(1000); // Wait 1 second
 }
 }
}

void SystemClock_Config(void)
{
 RCC_OscInitTypeDef RCC_OscInitStruct = {0};
 RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};
 RCC_PeriphCLKInitTypeDef PeriphClkInit = {0};

 RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
```

```

RCC_OscInitStruct.HSIState = RCC_HSI_ON;
RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
RCC_OscInitStruct.PLL.PLLM = 2;
RCC_OscInitStruct.PLL.PLLN = 20;
RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV7;
RCC_OscInitStruct.PLL.PLLQ = RCC_PLLQ_DIV2;
RCC_OscInitStruct.PLL.PLLR = RCC_PLLR_DIV2;
if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
{
 Error_Handler();
}

RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
 |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_4) != HAL_OK)
{
 Error_Handler();
}

PeriphClkInit.PeriphClockSelection = RCC_PERIPHCLK_ADC;
PeriphClkInit.AdcClockSelection = RCC_ADCCLKSOURCE_PLLSAI1;
PeriphClkInit.PLLSAI1.PLLSAI1Source = RCC_PLLSOURCE_HSI;
PeriphClkInit.PLLSAI1.PLLSAI1M = 2;
PeriphClkInit.PLLSAI1.PLLSAI1N = 8;
PeriphClkInit.PLLSAI1.PLLSAI1P = RCC_PLLP_DIV7;
PeriphClkInit.PLLSAI1.PLLSAI1Q = RCC_PLLQ_DIV2;
PeriphClkInit.PLLSAI1.PLLSAI1R = RCC_PLLR_DIV2;
PeriphClkInit.PLLSAI1.PLLSAI1ClockOut = RCC_PLLSAI1_ADC1CLK;
if (HAL_RCCEx_PeriphCLKConfig(&PeriphClkInit) != HAL_OK)
{
 Error_Handler();
}

if (HAL_PWREx_ControlVoltageScaling(PWR_REGULATOR_VOLTAGE_SCALE1) != HAL_OK)
{
 Error_Handler();
}
}

static void MX_ADC1_Init(void)

```

```
{
 ADC_MultiModeTypeDef multimode = {0};
 ADC_ChannelConfTypeDef sConfig = {0};

 hadc1.Instance = ADC1;
 hadc1.Init.ClockPrescaler = ADC_CLOCK_ASYNC_DIV1;
 hadc1.Init.Resolution = ADC_RESOLUTION_12B;
 hadc1.Init.DataAlign = ADC_DATAALIGN_RIGHT;
 hadc1.Init.ScanConvMode = ADC_SCAN_DISABLE;
 hadc1.Init.EOCSelection = ADC_EOC_SINGLE_CONV;
 hadc1.Init.LowPowerAutoWait = DISABLE;
 hadc1.Init.ContinuousConvMode = DISABLE;
 hadc1.Init.NbrOfConversion = 1;
 hadc1.Init.DiscontinuousConvMode = DISABLE;
 hadc1.Init.NbrOfDiscConversion = 1;
 hadc1.Init.ExternalTrigConv = ADC_SOFTWARE_START;
 hadc1.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_NONE;
 hadc1.Init.DMAContinuousRequests = DISABLE;
 hadc1.Init.Overrun = ADC_OVR_DATA_OVERWRITTEN;
 hadc1.Init.OversamplingMode = DISABLE;
 if (HAL_ADC_Init(&hadc1) != HAL_OK)
 {
 Error_Handler();
 }

 multimode.Mode = ADC_MODE_INDEPENDENT;
 if (HAL_ADCEx_MultiModeConfigChannel(&hadc1, &multimode) != HAL_OK)
 {
 Error_Handler();
 }

 sConfig.Channel = ADC_CHANNEL_3;
 sConfig.Rank = ADC_REGULAR_RANK_1;
 sConfig.SamplingTime = ADC_SAMPLETIME_2CYCLES_5;
 sConfig.SingleDiff = ADC_SINGLE_ENDED;
 sConfig.OffsetNumber = ADC_OFFSET_NONE;
 sConfig.Offset = 0;
 if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
 {
 Error_Handler();
 }
}

static void MX_GPIO_Init(void)
{
 GPIO_InitTypeDef GPIO_InitStruct = {0};
```

```

/* GPIO Ports Clock Enable */
__HAL_RCC_GPIOC_CLK_ENABLE();
__HAL_RCC_GPIOA_CLK_ENABLE();

/*Configure GPIO pin Output Level */
HAL_GPIO_WritePin(GPIOA, GPIO_PIN_6|GPIO_PIN_7|GPIO_PIN_8|GPIO_PIN_9
 |GPIO_PIN_10|GPIO_PIN_11, GPIO_PIN_RESET);

/*Configure GPIO pins : PA6 PA7 PA8 PA9
 PA10 PA11 */
GPIO_InitStruct.Pin = GPIO_PIN_6|GPIO_PIN_7|GPIO_PIN_8|GPIO_PIN_9
 |GPIO_PIN_10|GPIO_PIN_11;
GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
GPIO_InitStruct.Pull = GPIO_NOPULL;
GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);

}

void Error_Handler(void)
{

}

#ifdef USE_FULL_ASSERT

void assert_failed(uint8_t *file, uint32_t line)
{

}

#endif

/***** (C) COPYRIGHT STMicroelectronics *****/

```

*Figure 8.11: The ADCINT program.*

## 8.6 Project 4: Analogue Temperature Sensor

### Description

In this project an analogue temperature sensor is used to read the ambient temperature. The temperature is displayed on an LCD every second in degrees Centigrade in the following format

```

TEMPERATURE
nnnnn.nnnn

```

### The aim

The aim of this project is to show how an analogue input of the STM32L476 processor can be used to read the temperature from an analogue temperature sensor chip.

### Block diagram

The block diagram of the project is shown in Figure 8.12.

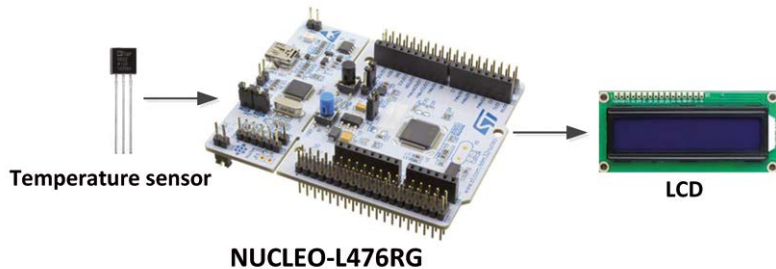


Figure 8.12: Block diagram of the project.

### Circuit diagram

The circuit diagram of the project is shown in Figure 8.13. A TMP36 type analogue temperature sensor is used in the project. This sensor can measure the temperature in the range  $-40\text{ }^{\circ}\text{C}$  to  $+125\text{ }^{\circ}\text{C}$  with an accuracy of  $\pm 2\text{ }^{\circ}\text{C}$ . The operating voltage of the sensor is  $+2.7\text{ V}$  to  $+5.5\text{ V}$ . The sensor output is an analogue voltage proportional to the measure temperature where the temperature is given by:

$$T = (V_o - 500) / 10$$

Where  $T$  is the measure temperature in degrees centigrade, and  $V_o$  is the sensor output voltage in millivolts. For example, if the voltage is  $600\text{ mV}$  then the temperature is  $10\text{ }^{\circ}\text{C}$ . Similarly, if the voltage is  $800\text{ mV}$  then the temperature is  $30\text{ }^{\circ}\text{C}$ . The output pin of the sensor chip is connected to analogue input PC2 of the Nucleo-L476RG board. The power input is connected to  $+3.3\text{ V}$  on the morpho connector. The LCD is powered from  $+5\text{ V}$  and is connected as in the previous project.

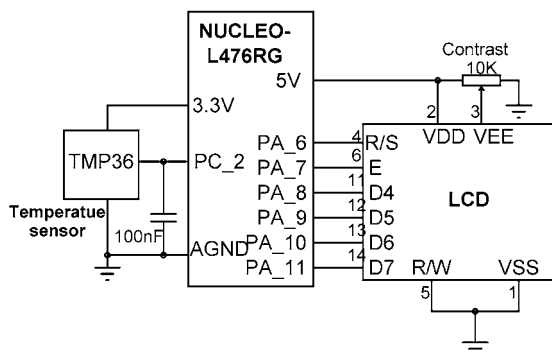


Figure 8.13: Circuit diagram of the project.

**Program listing**

The steps are as follows.

- Start STM32CubeIDE as before.
- Create a new workspace.
- Select the board type as Nucleo-L476RG.
- Give the name **TMP36** to the project.
- Configure PA6 to PA11 as digital output.
- Configure PC2 as ADC1\_IN3.
- Click the **Analog** tab and select **ADC1** and enable channel **IN3** as **Single-ended** to enable ADC on this channel (Pin PC2).
- Under **Parameter Settings**, click on **Resolution** and select **ADC 12-bit resolution**. Also, enable the **Continuous Conversion Mode**.
- Set **End of Conversion Selection** to **End of sequence of conversions**.
- Set **Overrun behaviour** to **Overrun data overwritten**.
- Click **Clock Configuration** and set clock to 80 MHz, make sure that the system clock and **ADC** clock are set as required.
- Click **File**, followed by **Save** and click **YES** to generate code.
- Click **Core**, followed by **Src** and double click **main.c** to open the main program.

You should now copy file **LCD.c** to the **Src** directory and **LCD.h** to **Inc** directory of your STM32CubeIDE working directory.

The following variable is declared to store the converted temperature data:

```
uint32_t adcResult = 0;
```

Inside the program loop the ADC is started, analogue data is read and stored in variable **adcResult**. This data is then converted into millivolts by multiplying with 3300 and dividing with 4095. This is because we have a 12-bit ADC, and the reference voltage is 3.3V, i.e. 0 corresponds to 0 V and 4095 corresponds to 3.3 V. The temperature is then calculated as a floating-point number by subtracting 500 and dividing by 10. The temperature is converted into a string by using function **sprintf**. The result is a string storing the number with two digits before and two digits after the decimal point. The temperature is finally displayed on the LCD. The above process is repeated every second.

The program loop contains the following statements:

```
while (1)
{
 HAL_ADC_PollForConversion(&hadc1, 100);
 adcResult = HAL_ADC_GetValue(&hadc1);
 Temperature = ((float)adcResult) * 3300.0 / 4095.0;
 Temperature = (Temperature - 500.0) / 10.0;
 lcd_goto(0, 1);
 sprintf(Buff, "%5.2f", Temperature);
 lcd_puts(Buff);
}
```



```
 HAL_Delay(1000);
 }
}
```

Compile the program in **Release** mode making sure there are no errors. Drag and drop the binary file TMP36.bin to device NUCLEO\_L476RG. Figure 8.14 shows the complete program listing (comments are removed for clarity).

```
/* USER CODE BEGIN Header */
/**

* @file : main.c
* @brief : Main program body

* @attention
*
* <h2><center>© Copyright (c) 2020 STMicroelectronics.
* All rights reserved.</center></h2>
*
* This software component is licensed by ST under BSD 3-Clause license,
* the «License»; You may not use this file except in compliance with the
* License. You may obtain a copy of the License at:
*
* opensource.org/licenses/BSD-3-Clause
*

*/
#include «main.h»
#include «LCD.h»
#include «stdio.h»

ADC_HandleTypeDef hadc1;

void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_ADC1_Init(void);

int main(void)
{
 char Buff[10];
 float Temperature;
 uint32_t adcResult;

 HAL_Init();

 SystemClock_Config();

 MX_GPIO_Init();

}
```

```

MX_ADC1_Init();
lcd_init(); // Init LCD
lcd_puts(«Temperature»); // Display heading
HAL_ADC_Start(&hadc1); // Start ADC

while (1)
{
 HAL_ADC_PollForConversion(&hadc1, 100); // Poll ADC
 adcResult = HAL_ADC_GetValue(&hadc1); // Get ADC value
 Temperature = ((float)adcResult) * 3300.0 / 4095.0; // Result in mV
 Temperature = (Temperature - 500.0) / 10.0; // Result in Centigrade
 lcd_goto(0, 1); // LCD col 0, row 1
 sprintf(Buff, «%5.2f», Temperature); // Convert to string
 lcd_puts(Buff); // Display temperature
 HAL_Delay(1000); // Wait 1 second
}
}

void SystemClock_Config(void)
{
 RCC_OscInitTypeDef RCC_OscInitStruct = {0};
 RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};
 RCC_PeriphCLKInitTypeDef PeriphClkInit = {0};

 RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
 RCC_OscInitStruct.HSISState = RCC_HSI_ON;
 RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
 RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
 RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
 RCC_OscInitStruct.PLL.PLLM = 2;
 RCC_OscInitStruct.PLL.PLLN = 20;
 RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV7;
 RCC_OscInitStruct.PLL.PLLQ = RCC_PLLQ_DIV2;
 RCC_OscInitStruct.PLL.PLLR = RCC_PLLR_DIV2;
 if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
 {
 Error_Handler();
 }

 RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
 |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
 RCC_ClkInitStruct.SYSClkSource = RCC_SYSCCLKSOURCE_PLLCLK;
 RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCCLK_DIV1;
 RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
 RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

```

```
if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_4) != HAL_OK)
{
 Error_Handler();
}
PeriphClkInit.PeriphClockSelection = RCC_PERIPHCLK_ADC;
PeriphClkInit.AdcClockSelection = RCC_ADCCLKSOURCE_PLLSAI1;
PeriphClkInit.PLLSAI1.PLLSAI1Source = RCC_PLLSOURCE_HSI;
PeriphClkInit.PLLSAI1.PLLSAI1M = 2;
PeriphClkInit.PLLSAI1.PLLSAI1N = 8;
PeriphClkInit.PLLSAI1.PLLSAI1P = RCC_PLLP_DIV7;
PeriphClkInit.PLLSAI1.PLLSAI1Q = RCC_PLLQ_DIV2;
PeriphClkInit.PLLSAI1.PLLSAI1R = RCC_PLLR_DIV2;
PeriphClkInit.PLLSAI1.PLLSAI1ClockOut = RCC_PLLSAI1_ADC1CLK;
if (HAL_RCCEx_PeriphCLKConfig(&PeriphClkInit) != HAL_OK)
{
 Error_Handler();
}

if (HAL_PWREx_ControlVoltageScaling(PWR_REGULATOR_VOLTAGE_SCALE1) != HAL_OK)
{
 Error_Handler();
}
}

static void MX_ADC1_Init(void)
{
 ADC_MultiModeTypeDef multimode = {0};
 ADC_ChannelConfTypeDef sConfig = {0};

 hadc1.Instance = ADC1;
 hadc1.Init.ClockPrescaler = ADC_CLOCK_ASYNC_DIV1;
 hadc1.Init.Resolution = ADC_RESOLUTION_12B;
 hadc1.Init.DataAlign = ADC_DATAALIGN_RIGHT;
 hadc1.Init.ScanConvMode = ADC_SCAN_DISABLE;
 hadc1.Init.EOCSelection = ADC_EOC_SINGLE_CONV;
 hadc1.Init.LowPowerAutoWait = DISABLE;
 hadc1.Init.ContinuousConvMode = ENABLE;
 hadc1.Init.NbrOfConversion = 1;
 hadc1.Init.DiscontinuousConvMode = DISABLE;
 hadc1.Init.NbrOfDiscConversion = 1;
 hadc1.Init.ExternalTrigConv = ADC_SOFTWARE_START;
 hadc1.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_NONE;
 hadc1.Init.DMAContinuousRequests = DISABLE;
 hadc1.Init.Overrun = ADC_OVR_DATA_OVERWRITTEN;
 hadc1.Init.OversamplingMode = DISABLE;
 if (HAL_ADC_Init(&hadc1) != HAL_OK)
```

```
{
 Error_Handler();
}

multimode.Mode = ADC_MODE_INDEPENDENT;
if (HAL_ADCEx_MultiModeConfigChannel(&hadc1, &multimode) != HAL_OK)
{
 Error_Handler();
}

sConfig.Channel = ADC_CHANNEL_3;
sConfig.Rank = ADC_REGULAR_RANK_1;
sConfig.SamplingTime = ADC_SAMPLETIME_2CYCLES_5;
sConfig.SingleDiff = ADC_SINGLE_ENDED;
sConfig.OffsetNumber = ADC_OFFSET_NONE;
sConfig.Offset = 0;
if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
{
 Error_Handler();
}
}

static void MX_GPIO_Init(void)
{
 GPIO_InitTypeDef GPIO_InitStruct = {0};

 /* GPIO Ports Clock Enable */
 __HAL_RCC_GPIOC_CLK_ENABLE();
 __HAL_RCC_GPIOA_CLK_ENABLE();

 /*Configure GPIO pin Output Level */
 HAL_GPIO_WritePin(GPIOA, GPIO_PIN_6|GPIO_PIN_7|GPIO_PIN_8|GPIO_PIN_9
 |GPIO_PIN_10|GPIO_PIN_11, GPIO_PIN_RESET);

 /*Configure GPIO pins : PA6 PA7 PA8 PA9
 PA10 PA11 */
 GPIO_InitStruct.Pin = GPIO_PIN_6|GPIO_PIN_7|GPIO_PIN_8|GPIO_PIN_9
 |GPIO_PIN_10|GPIO_PIN_11;
 GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
 GPIO_InitStruct.Pull = GPIO_NOPULL;
 GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
 HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
}

void Error_Handler(void)
```

```
{
}

#ifdef USE_FULL_ASSERT

void assert_failed(uint8_t *file, uint32_t line)
{

}
#endif

/***** (C) COPYRIGHT STMicroelectronics *****/
```

*Figure 8.14: The TMP36 program.*

## 8.7 Project 5: ON-OFF Temperature Controller

### Description

This is a simple ON-OFF temperature control system. The system consists of an analogue temperature sensor chip and a heater, connected to a relay. The temperature sensor measures the oven temperature and compares with a preset value. If the measured temperature is lower than the preset value then the heater is turned ON. If on the other hand the measured temperature is higher than the preset value, then the heater is turned OFF.

The preset and the measured temperatures are both displayed on the LCD in the following format:

```
Set: nn.nn
Meas: nn.nn
```

Where **Set** is the setpoint temperature, and **Meas** is the measured temperature.

### The aim

The aim of this project is to show how a simple ON-OFF temperature controller can be developed.

### Block diagram

The block diagram of the project is shown in Figure 8.15.

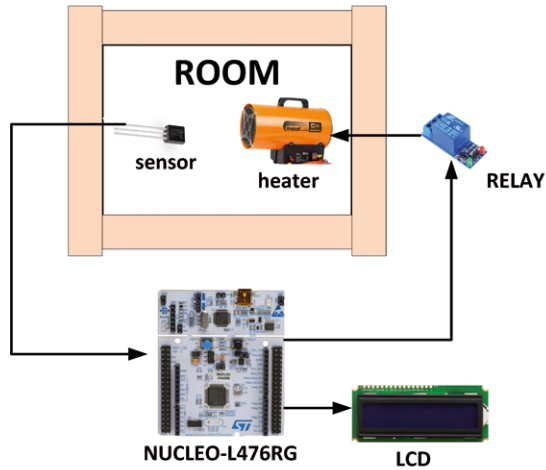


Figure 8.15: Block diagram of the project.

### Circuit diagram

Figure 8.16 shows the circuit diagram of the project. A TMP36 temperature sensor chip is used as in the previous project and is connected to PC2. The heater is connected to port pin PC1 through a relay. The LCD is connected to the system as in the previous project.

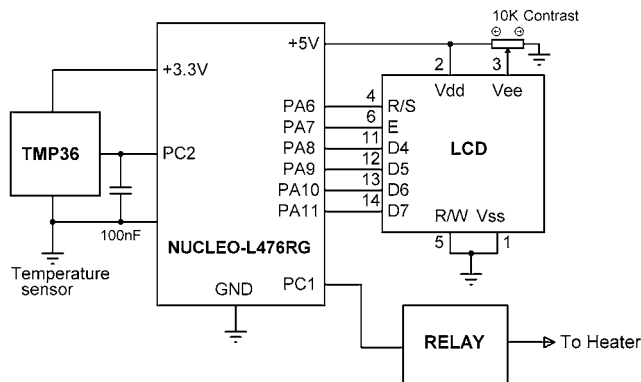


Figure 8.16: Circuit diagram of the project.

### Program listing

The steps are as follows.

- Start STM32CubeIDE as before.
- Create a new workspace.
- Select the board type as Nucleo-L476RG.
- Give the name **ONOFF** to the project.
- Configure PA6 to PA11 as digital output.
- Configure PC2 as ADC1\_IN3.
- Click the **Analog** tab and select **ADC1** and enable channel **IN3** as **Single-ended** to enable ADC on this channel (Pin PC2).

- Under **Parameter Settings**, click on **Resolution** and select **ADC 12-bit resolution**. Also, enable the **Continuous Conversion Mode**.
- Set **End of Conversion Selection** to **End of single conversion**.
- Set **Overrun behaviour** to **Overrun data overwritten**.
- Click **Clock Configuration** and set clock to 80 MHz, make sure that the system clock and **ADC** clock are set as required.
- Click **File**, followed by **Save** and click **YES** to generate code.
- Click **Core**, followed by **Src** and double click **main.c** to open the main program.

You should now copy file **LCD.c** to the **Src** directory and **LCD.h** to **Inc** directory of your STM32CubeIDE working directory.

Two functions are defined in the program: **HeaterON** and **HeaterOFF**. The setpoint temperature is set to 25.0 °C and is stored in floating point variable SetPoint.

The operation of the program can be described by the following PDL:

```
BEGIN
 Initialize setpoint temperature
 Initialize LCD
 Turn OFF heater
 Start ADC
 DO FOREVER
 Display setpoint temperature at top row of the LCD
 Read analogue temperature from ADC
 Convert to °C
 Display temperature at bottom row of LCD
 IF setpoint > temperature THEN
 Turn heater OFF
 ELSE
 Turn heater ON
 ENDIF
 Wait one second
 Clear LCD
 ENDDO
END
```

The code to implement the ON-OFF temperature control algorithm is as follows.

```
while (1)
{
 char Setp[20] = " Set: ";
 char Meas[20] = "Meas: ";

 strcat(Setp, BuffSet);

 lcd_goto(0,0); // Top row
 lcd_puts(Setp); // Disp setpoint
```

```

 HAL_ADC_PollForConversion(&hadc1, 100); // Poll ADC
 adcResult = HAL_ADC_GetValue(&hadc1); // Get ADC value
 Temperature = ((float)adcResult) * 3300.0 / 4095.0; // Result in mV
 Temperature = (Temperature - 500.0) / 10.0; // Result in C
 lcd_goto(0, 1); // Bottom row
 sprintf(BuffMeasure, "%5.2f", Temperature); // To string
 strcat(Meas, BuffMeasure);
 lcd_puts(Meas); // Disp temp

 if(SetPoint > Temperature)
 HeaterOFF(); // Heater OFF
 else
 HeaterON(); // Heater ON

 HAL_Delay(1000); // Wait 1 second
 lcd_clear(); // Clear LCD
 }

```

Assuming the setpoint temperature is 25.00 degrees and the measured temperature is 28.50 degrees, the LCD will display the temperatures in the following format:

```

Set: 25.00
Meas: 28.50

```

Figure 8.17 shows the complete program listing (the comments have been removed for clarity).

```

/* USER CODE BEGIN Header */
/**
 *
 * @file : main.c
 * @brief : Main program body
 *
 * @attention
 *
 * <h2><center>© Copyright (c) 2020 STMicroelectronics.
 * All rights reserved.</center></h2>
 *
 * This software component is licensed by ST under BSD 3-Clause license,
 * the «License»; You may not use this file except in compliance with the
 * License. You may obtain a copy of the License at:
 *
 * opensource.org/licenses/BSD-3-Clause
 *
 */
#include «main.h»

```



```
#include «LCD.h»
#include «stdio.h»
#include «string.h»

#define heater GPIO_PIN_1

ADC_HandleTypeDef hadc1;

void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_ADC1_Init(void);

//
// This function turns OFF the heater
//
void HeaterOFF()
{
 HAL_GPIO_WritePin(GPIOC, heater, GPIO_PIN_RESET);
}

//
// This function turns ON the heater
//
void HeaterON()
{
 HAL_GPIO_WritePin(GPIOC, heater, GPIO_PIN_SET);
}

//
// Start of main program
//
int main(void)
{
 char BuffMeasure[20], BuffSet[20];
 float Temperature;
 float SetPoint = 25.0; // Set-point
 uint32_t adcResult;
 HAL_Init();

 SystemClock_Config();

 MX_GPIO_Init();
 MX_ADC1_Init();
 lcd_init(); // Init LCD
 HeaterOFF(); // heater OFF
 HAL_ADC_Start(&hadc1); // Start ADC
```

```

sprintf(BuffSet, «%5.2f», SetPoint); // To string

while (1)
{
 char Setp[20] = « Set: «;
 char Meas[20] = «Meas: «;

 strcat(Setp, BuffSet);
 lcd_goto(0,0); // Top row
 lcd_puts(Setp); // Display setpoint

 HAL_ADC_PollForConversion(&hadc1, 100); // Poll ADC
 adcResult = HAL_ADC_GetValue(&hadc1); // Get ADC value
 Temperature = ((float)adcResult) * 3300.0 / 4095.0; // Result in mV
 Temperature = (Temperature - 500.0) / 10.0; // Result in C
 lcd_goto(0, 1); // Bottom row
 sprintf(BuffMeasure, «%5.2f», Temperature); // Convert to string
 strcat(Meas, BuffMeasure);
 lcd_puts(Meas); // Display temperature

 if(SetPoint > Temperature) // SetPoint greater?
 HeaterOFF(); // Heater OFF
 else
 HeaterON(); // Heater ON

 HAL_Delay(1000); // Wait 1 second
 lcd_clear(); // Clear LCD
}
}

void SystemClock_Config(void)
{
 RCC_OscInitTypeDef RCC_OscInitStruct = {0};
 RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};
 RCC_PeriphCLKInitTypeDef PeriphClkInit = {0};

 RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
 RCC_OscInitStruct.HSISState = RCC_HSI_ON;
 RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
 RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
 RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
 RCC_OscInitStruct.PLL.PLLM = 2;
 RCC_OscInitStruct.PLL.PLLN = 20;
 RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV7;
 RCC_OscInitStruct.PLL.PLLQ = RCC_PLLQ_DIV2;
 RCC_OscInitStruct.PLL.PLLR = RCC_PLLR_DIV2;

```

```
if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
{
 Error_Handler();
}

RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
 |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_4) != HAL_OK)
{
 Error_Handler();
}
PeriphClkInit.PeriphClockSelection = RCC_PERIPHCLK_ADC;
PeriphClkInit.AdcClockSelection = RCC_ADCCLKSOURCE_PLLSAI1;
PeriphClkInit.PLLSAI1.PLLSAI1Source = RCC_PLLSOURCE_HSI;
PeriphClkInit.PLLSAI1.PLLSAI1M = 2;
PeriphClkInit.PLLSAI1.PLLSAI1N = 8;
PeriphClkInit.PLLSAI1.PLLSAI1P = RCC_PLLP_DIV7;
PeriphClkInit.PLLSAI1.PLLSAI1Q = RCC_PLLQ_DIV2;
PeriphClkInit.PLLSAI1.PLLSAI1R = RCC_PLLR_DIV2;
PeriphClkInit.PLLSAI1.PLLSAI1ClockOut = RCC_PLLSAI1_ADC1CLK;
if (HAL_RCCEx_PeriphCLKConfig(&PeriphClkInit) != HAL_OK)
{
 Error_Handler();
}

if (HAL_PWREx_ControlVoltageScaling(PWR_REGULATOR_VOLTAGE_SCALE1) != HAL_OK)
{
 Error_Handler();
}
}

static void MX_ADC1_Init(void)
{
 ADC_MultiModeTypeDef multimode = {0};
 ADC_ChannelConfTypeDef sConfig = {0};

 hadc1.Instance = ADC1;
 hadc1.Init.ClockPrescaler = ADC_CLOCK_ASYNC_DIV1;
 hadc1.Init.Resolution = ADC_RESOLUTION_12B;
 hadc1.Init.DataAlign = ADC_DATAALIGN_RIGHT;
 hadc1.Init.ScanConvMode = ADC_SCAN_DISABLE;
```

```

hadc1.Init.EOCSelection = ADC_EOC_SINGLE_CONV;
hadc1.Init.LowPowerAutoWait = DISABLE;
hadc1.Init.ContinuousConvMode = ENABLE;
hadc1.Init.NbrOfConversion = 1;
hadc1.Init.DiscontinuousConvMode = DISABLE;
hadc1.Init.NbrOfDiscConversion = 1;
hadc1.Init.ExternalTrigConv = ADC_SOFTWARE_START;
hadc1.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_NONE;
hadc1.Init.DMAContinuousRequests = DISABLE;
hadc1.Init.Overrun = ADC_OVR_DATA_OVERWRITTEN;
hadc1.Init.OversamplingMode = DISABLE;
if (HAL_ADC_Init(&hadc1) != HAL_OK)
{
 Error_Handler();
}

multimode.Mode = ADC_MODE_INDEPENDENT;
if (HAL_ADCEx_MultiModeConfigChannel(&hadc1, &multimode) != HAL_OK)
{
 Error_Handler();
}

sConfig.Channel = ADC_CHANNEL_3;
sConfig.Rank = ADC_REGULAR_RANK_1;
sConfig.SamplingTime = ADC_SAMPLETIME_2CYCLES_5;
sConfig.SingleDiff = ADC_SINGLE_ENDED;
sConfig.OffsetNumber = ADC_OFFSET_NONE;
sConfig.Offset = 0;
if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
{
 Error_Handler();
}
}

static void MX_GPIO_Init(void)
{
 GPIO_InitTypeDef GPIO_InitStruct = {0};

 /* GPIO Ports Clock Enable */
 __HAL_RCC_GPIOC_CLK_ENABLE();
 __HAL_RCC_GPIOA_CLK_ENABLE();

 /*Configure GPIO pin Output Level */
 HAL_GPIO_WritePin(GPIOC, GPIO_PIN_1, GPIO_PIN_RESET);

 /*Configure GPIO pin Output Level */

```

```
HAL_GPIO_WritePin(GPIOA, GPIO_PIN_6|GPIO_PIN_7|GPIO_PIN_8|GPIO_PIN_9
 |GPIO_PIN_10|GPIO_PIN_11, GPIO_PIN_RESET);

/*Configure GPIO pin : PC1 */
GPIO_InitStruct.Pin = GPIO_PIN_1;
GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
GPIO_InitStruct.Pull = GPIO_NOPULL;
GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);

/*Configure GPIO pins : PA6 PA7 PA8 PA9
 PA10 PA11 */
GPIO_InitStruct.Pin = GPIO_PIN_6|GPIO_PIN_7|GPIO_PIN_8|GPIO_PIN_9
 |GPIO_PIN_10|GPIO_PIN_11;
GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
GPIO_InitStruct.Pull = GPIO_NOPULL;
GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);

}

void Error_Handler(void)
{
}

#ifdef USE_FULL_ASSERT

void assert_failed(uint8_t *file, uint32_t line)
{
}

#endif

/***** (C) COPYRIGHT STMicroelectronics *****/
```

*Figure 8.17: The ONOFF program.*

## 8.8 Project 6: Multiple-input ADC with DMA

ADC operation in DMA mode is very useful as it allows multiple channel data to be read easily and the results to be stored automatically in memory without the intervention of the CPU. Setting the DMA in circular mode allows the conversions to be performed continuously.

### Description

In this section we will use two analogue inputs PC2 and PC3 to measure two analogue input voltages using DMA based ADC. The measured voltages are displayed on the top and bottom rows of the LCD in millivolts.

### The aim

The aim of this project is to show how DMA can be used to fetch analogue data from two channels.

### Circuit diagram

The circuit diagram is same as in Figure 8.7.

### Program listing

The steps are as follows:

- Start STM32CubeIDE as before.
- Create a new workspace.
- Select the board type as Nucleo-L476RG.
- Give the name **ADCDMA** to the project.
- Configure PA6 to PA11 as digital outputs for the LCD.
- Configure PC2 as ADC1\_IN3.
- Configure PC3 as ADC1\_IN4.
- Click the **Analog** tab and select **ADC1** and enable channel **IN3** as **Single-ended** to enable ADC on this channel (Pin PC2). Similarly, enable **IN4** as **Single-ended** to enable ADC on this channel (PC3).
- Click **DMA** under tab **System Core** and select **ADC1** as shown in Figure 8.18.

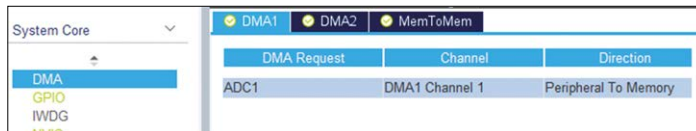


Figure 8.18: Selecting ADC1 in DMA.

- Set **Mode** to **Normal** and **Data Width** to **Half Word** (Figure 8.19)

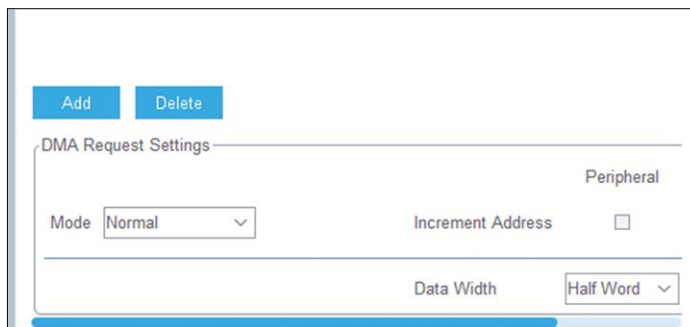


Figure 8.19: Selecting DMA mode and data width.

- Click the **Analog** tab and select **ADC1**. Under **Parameter Settings**, click on **Resolution** and select **ADC 12-bit resolution**. Enable **Continuous Conversion Mode**.
- Enable **Scan Conversion Mode** and **Continuous Conversion Mode**.
- Enable **DMA Continuous Requests**.

- Set **End of Conversion Selection** to **End of sequence of conversion**.
- Set **Overrun behaviour** to **Overrun data overwritten**.
- Set **Number of Conversions** to **2**.
- Set **Rank 1 Channel** to **Channel 3**. Also, set **Rank 2 Channel** to **Channel 4** (Figure 8.20).

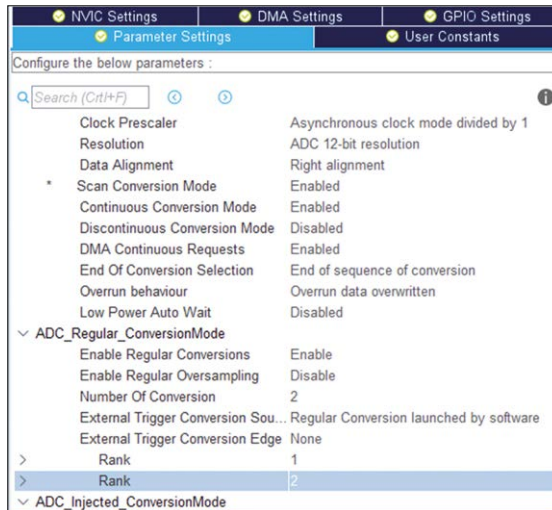


Figure 8.20: Parameter settings.

- Set the MCU clock to 80 MHz.
- Click **File**, followed by **Save** and click **YES** to generate code.
- Click **Core**, followed by **Src** and double click **main.c** to open the main program.

You should now copy file **LCD.c** to the **Src** directory and **LCD.h** to **Inc** directory of your STM32CubeIDE working directory. Include files **LCD.h** and **stdio.h** at the beginning of the program. Define variable **adcBuffer** which will be loaded with the converted data by the DMA. Also define a variable called **convCompleted** and initialize it to 0. This variable will be set to one when the conversion is complete, and the converted data has been loaded to the specified buffer.

```
uint16_t adcBuffer[2]; // DMA buffer
volatile uint8_t convCompleted = 0; // Convert flag
```

As in the interrupt-based ADC, the callback function is named as:

```
HAL_ADC_ConvCpltCallback
```

Inside the main program, before the program loop, we must define the following function to start the DMA based ADC. the LCD is also initialized here:

```
HAL_ADC_Start_DMA(&hadc1, (uint32_t*)adcBuffer, 2);
```

The program loop is active when variable **convCompleted** becomes 1. Here, the ADC is stopped by statement **HAL\_ADC\_Stop\_DMA(&hadc1)**. In the remaining parts of the program loop, we extract the converted data from buffer **adcBuffer[0]**, convert it to millivolts, and then display at the top row of the LCD. The data from the second analogue input is extracted from **adcBuffer[1]** and displayed at the bottom row of the LCD. The program then enables the DMA, and the above process gets repeated. Notice that the ADC DMA is stopped and then re-started inside the program loop.

The measured voltages are displayed on the LCD in the following format:

```
nn.nn
nn.nn
```

Figure 8.21 shows the program listing. Compile the program in **Release** mode, drag and drop the binary file **ADCDMA.bin** to device NUCLEO\_L476RG.

```
/* USER CODE BEGIN Header */
/**
 * *****
 * @file : main.c
 * @brief : Main program body
 * *****
 * @attention
 *
 * <h2><center>© Copyright (c) 2020 STMicroelectronics.
 * All rights reserved.</center></h2>
 *
 * This software component is licensed by ST under BSD 3-Clause license,
 * the «License»; You may not use this file except in compliance with the
 * License. You may obtain a copy of the License at:
 *
 * opensource.org/licenses/BSD-3-Clause
 *
 * *****
 */
#include «main.h»
#include «LCD.h»
#include «stdio.h»

ADC_HandleTypeDef hadc1;
DMA_HandleTypeDef hdma_adc1;

void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_DMA_Init(void);
static void MX_ADC1_Init(void);

uint16_t adcBuffer[2]; // DMA buffer
```



```
volatile uint8_t convCompleted = 0; // Convert flag

//
// End of conversion function
//
void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef* hadc1)
{
 convCompleted = 1;
}

//
// Start of main program
//
int main(void)
{
 char Buff[10];
 float mv;

 HAL_Init();

 SystemClock_Config();

 MX_GPIO_Init();
 MX_DMA_Init();
 MX_ADC1_Init();
 HAL_ADC_Start_DMA(&hadc1, (uint32_t*)adcBuffer, 2); // Start ADC in DMA
 lcd_init(); // Init LCD

 while (1)
 {
 while(!convCompleted); // Wait until completed
 HAL_ADC_Stop_DMA(&hadc1); // Stop ADC DMA
 convCompleted = 0;

 mv = ((float)adcBuffer[0]) * 3300.0 / 4095.0; // Result in mV
 lcd_goto(0, 0); // LCD top row
 sprintf(Buff, «%5.2f», mv); // Convert to string
 lcd_puts(Buff); // Display temperature

 mv = ((float)adcBuffer[1]) * 3300.0 / 4095.0; // Result in mV
 lcd_goto(0, 1); // LCD bottom row
 sprintf(Buff, «%5.2f», mv); // Convert to string
 lcd_puts(Buff); // Display temperature
 HAL_Delay(1000); // Wait 1 second

 HAL_ADC_Start_DMA(&hadc1, (uint32_t*)adcBuffer, 2); // Start ADC in DMA
 }
}
```

```

 lcd_clear(); // Clear LCD
 }
}

/**
 * @brief System Clock Configuration
 * @retval None
 */
void SystemClock_Config(void)
{
 RCC_OscInitTypeDef RCC_OscInitStruct = {0};
 RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};
 RCC_PeriphCLKInitTypeDef PeriphClkInit = {0};

 /** Initializes the CPU, AHB and APB busses clocks
 */
 RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
 RCC_OscInitStruct.HSISState = RCC_HSI_ON;
 RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
 RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
 RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
 RCC_OscInitStruct.PLL.PLLM = 2;
 RCC_OscInitStruct.PLL.PLLN = 20;
 RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV7;
 RCC_OscInitStruct.PLL.PLLQ = RCC_PLLQ_DIV2;
 RCC_OscInitStruct.PLL.PLLR = RCC_PLLR_DIV2;
 if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
 {
 Error_Handler();
 }
 /** Initializes the CPU, AHB and APB busses clocks
 */
 RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
 |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
 RCC_ClkInitStruct.SYSClkSource = RCC_SYSCCLKSOURCE_PLLCLK;
 RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCCLK_DIV1;
 RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
 RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

 if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_4) != HAL_OK)
 {
 Error_Handler();
 }
 PeriphClkInit.PeriphClockSelection = RCC_PERIPHCLK_ADC;
 PeriphClkInit.AdcClockSelection = RCC_ADCCCLKSOURCE_PLLSAI1;
 PeriphClkInit.PLLSAI1.PLLSAI1Source = RCC_PLLSOURCE_HSI;

```

```
PeriphClkInit.PLLSAI1.PLLSAI1M = 2;
PeriphClkInit.PLLSAI1.PLLSAI1N = 8;
PeriphClkInit.PLLSAI1.PLLSAI1P = RCC_PLLP_DIV7;
PeriphClkInit.PLLSAI1.PLLSAI1Q = RCC_PLLQ_DIV2;
PeriphClkInit.PLLSAI1.PLLSAI1R = RCC_PLLR_DIV2;
PeriphClkInit.PLLSAI1.PLLSAI1ClockOut = RCC_PLLSAI1_ADC1CLK;
if (HAL_RCCEx_PeriphCLKConfig(&PeriphClkInit) != HAL_OK)
{
 Error_Handler();
}
/** Configure the main internal regulator output voltage
 */
if (HAL_PWREx_ControlVoltageScaling(PWR_REGULATOR_VOLTAGE_SCALE1) != HAL_OK)
{
 Error_Handler();
}
}

/**
 * @brief ADC1 Initialization Function
 * @param None
 * @retval None
 */
static void MX_ADC1_Init(void)
{

 /* USER CODE BEGIN ADC1_Init 0 */

 /* USER CODE END ADC1_Init 0 */

 ADC_MultiModeTypeDef multimode = {0};
 ADC_ChannelConfTypeDef sConfig = {0};

 /* USER CODE BEGIN ADC1_Init 1 */

 /* USER CODE END ADC1_Init 1 */
 /** Common config
 */
 hadc1.Instance = ADC1;
 hadc1.Init.ClockPrescaler = ADC_CLOCK_ASYNC_DIV1;
 hadc1.Init.Resolution = ADC_RESOLUTION_12B;
 hadc1.Init.DataAlign = ADC_DATAALIGN_RIGHT;
 hadc1.Init.ScanConvMode = ADC_SCAN_ENABLE;
 hadc1.Init.EOCSelection = ADC_EOC_SEQ_CONV;
 hadc1.Init.LowPowerAutoWait = DISABLE;
 hadc1.Init.ContinuousConvMode = ENABLE;
```

```

hadc1.Init.NbrOfConversion = 2;
hadc1.Init.DiscontinuousConvMode = DISABLE;
hadc1.Init.NbrOfDiscConversion = 1;
hadc1.Init.ExternalTrigConv = ADC_SOFTWARE_START;
hadc1.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_NONE;
hadc1.Init.DMAContinuousRequests = ENABLE;
hadc1.Init.Overrun = ADC_OVR_DATA_OVERWRITTEN;
hadc1.Init.OversamplingMode = DISABLE;
if (HAL_ADC_Init(&hadc1) != HAL_OK)
{
 Error_Handler();
}
/** Configure the ADC multi-mode
*/
multimode.Mode = ADC_MODE_INDEPENDENT;
if (HAL_ADCEx_MultiModeConfigChannel(&hadc1, &multimode) != HAL_OK)
{
 Error_Handler();
}
/** Configure Regular Channel
*/
sConfig.Channel = ADC_CHANNEL_3;
sConfig.Rank = ADC_REGULAR_RANK_1;
sConfig.SamplingTime = ADC_SAMPLETIME_2CYCLES_5;
sConfig.SingleDiff = ADC_SINGLE_ENDED;
sConfig.OffsetNumber = ADC_OFFSET_NONE;
sConfig.Offset = 0;
if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
{
 Error_Handler();
}
/** Configure Regular Channel
*/
sConfig.Channel = ADC_CHANNEL_4;
sConfig.Rank = ADC_REGULAR_RANK_2;
if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
{
 Error_Handler();
}
/* USER CODE BEGIN ADC1_Init 2 */

/* USER CODE END ADC1_Init 2 */

}

/**

```

```
* Enable DMA controller clock
*/
static void MX_DMA_Init(void)
{

 /* DMA controller clock enable */
 __HAL_RCC_DMA1_CLK_ENABLE();

 /* DMA interrupt init */
 /* DMA1_Channel1_IRQn interrupt configuration */
 HAL_NVIC_SetPriority(DMA1_Channel1_IRQn, 0, 0);
 HAL_NVIC_EnableIRQ(DMA1_Channel1_IRQn);

}

/**
 * @brief GPIO Initialization Function
 * @param None
 * @retval None
 */
static void MX_GPIO_Init(void)
{
 GPIO_InitTypeDef GPIO_InitStruct = {0};

 /* GPIO Ports Clock Enable */
 __HAL_RCC_GPIOC_CLK_ENABLE();
 __HAL_RCC_GPIOA_CLK_ENABLE();

 /*Configure GPIO pin Output Level */
 HAL_GPIO_WritePin(GPIOA, GPIO_PIN_6|GPIO_PIN_7|GPIO_PIN_8|GPIO_PIN_9
 |GPIO_PIN_10|GPIO_PIN_11, GPIO_PIN_RESET);

 /*Configure GPIO pins : PA6 PA7 PA8 PA9
 PA10 PA11 */
 GPIO_InitStruct.Pin = GPIO_PIN_6|GPIO_PIN_7|GPIO_PIN_8|GPIO_PIN_9
 |GPIO_PIN_10|GPIO_PIN_11;
 GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
 GPIO_InitStruct.Pull = GPIO_NOPULL;
 GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
 HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);

}

/* USER CODE BEGIN 4 */

/* USER CODE END 4 */
```

```

/**
 * @brief This function is executed in case of error occurrence.
 * @retval None
 */
void Error_Handler(void)
{
 /* USER CODE BEGIN Error_Handler_Debug */
 /* User can add his own implementation to report the HAL error return state */

 /* USER CODE END Error_Handler_Debug */
}

#ifdef USE_FULL_ASSERT
/**
 * @brief Reports the name of the source file and the source line number
 * where the assert_param error has occurred.
 * @param file: pointer to the source file name
 * @param line: assert_param error line source number
 * @retval None
 */
void assert_failed(uint8_t *file, uint32_t line)
{
 /* USER CODE BEGIN 6 */
 /* User can add his own implementation to report the file name and line number,
 tex: printf(«Wrong parameters value: file %s on line %d\r\n», file, line) */
 /* USER CODE END 6 */
}
#endif /* USE_FULL_ASSERT */

/***** (C) COPYRIGHT STMicroelectronics *****/

```

*Figure 8.21: The ADCDMA program.*

### Modified program

In the program in Figure 8.21, we have to stop and then re-start the ADC DMA. We can modify the program so that the conversion is continuous. This requires the following to be set in the **Parameter Settings**:

- Enable **ContinuousConvMode**
- Enable **DMAContinuousRequests**

Also, set the **Mode** to **Circular** in DMA settings (under tab **System Core**). In circular mode of operation, a large buffer is usually specified which is filled automatically and continually by the DMA (the buffer pointer is incremented automatically). After the buffer is full, the callback is actioned and the buffer starts to fill again from the beginning. Note that two callback functions are supported in **Circular** mode:

- `HAL_ADC_ConvHalfCpltCallback()`: this is called when the buffer is half full.
- `HAL_ADC_ConvCpltCallback()`: this is called when the buffer is full.

### 8.9 Timer-driven ADC

The ADC can be driven from a timer through the **TRGO** trigger line. This feature is useful as it enables the ADC to perform conversions at a given frequency. For example, in digital signal processing applications we want to trigger the ADC at a fixed frequency. Both interrupt and DMA mode can be used with the timer mode. At high frequencies it is recommended to use the scan mode of conversion with the timer and DMA.

### 8.10 External-driven ADC

It is possible to configure an EXTI line to trigger ADC conversions. For example, in most MCUs EXTI line 11 can be enabled for external ADC operations. It is not possible to operate both in timer and in external trigger modes.

### 8.11 ADC calibration

The ADC on the STM32L476RG processor can be calibrated in software where a calibration factor is calculated and applied internally to the chip while the chip is up and running. The calibration process removes any chip to chip errors. If the calibration process is to be used, then it is important that the program waits until this process is complete before using the ADC to read external data.

The following HAL functions are available to calibrate the ADC:

**HAL\_ADCEX\_Calibration\_Start(handle, mode):** this function, called after the `HAL_ADC_Init()`, performs a calibration. The mode can be `ADC_SINGLE_ENDED` or `ADC_DIFFERENTIAL_ENDED`. Note that the calibration is applied automatically.

**HAL\_ADCEX\_Calibration\_GetValue(handle, mode):** this function reads the calibration value calculated by the above function. Mode takes the same values as above. Note that there is no need to retrieve the calibration value and apply it to the chip as it is done automatically.

**HAL\_ADCEX\_Calibration\_SetValue(handle, mode, CalibrationFactor):** is used to set up a custom derived calibration value.

### 8.12 Summary

In this Chapter we have developed several projects using the ADC of the Nucleo-L476RG development board. In the next Chapter we will be learning how to use the digital-to-analogue (DAC) converter in various projects.

## CHAPTER 9 • Using the Digital-to-Analogue Converters

### 9.1 Overview

In the previous Chapter we have focused our attention on the ADC module, showing its features. Several example projects are given to show how to use the ADC in projects with the STM32CubeIDE.

This Chapter gives a quick introduction to the digital-to-analogue converters (DAC) and describes with projects how to use the DAC on the Nucleo-L476RG development board together with the STM32CubeIDE software.

A DAC is a device that converts digital data to an analogue signal, proportional to a reference voltage. The DAC module on STM32 microcontrollers is based on the well-known  $R$ - $2R$  ladder network, which consists of high precision repetitive resistor chain.

The analogue output voltage  $V$  of an ADC is given by the following equation:

$$V = \frac{V_{ref} \times D}{2^N}$$

Where  $D$  is the digital input,  $N$  is the data width of the ADC, and  $V_{ref}$  is the reference voltage. For example, with a 12-bit ADC, having +3.3 V reference voltage and 12-bit digital input 0000 0010 0000 (i.e. 32), the output voltage will be:

$$V = \frac{3.3 \times 32}{2^{12}} = \frac{105.6}{4096} = 25.7 \text{ mV}$$

There are two 12-bit DAC on the Nucleo-L476RG development board, available at port pins PA4 and PA5. The DAC module incorporates a Sample & Hold unit for low-power applications. The DAC hardware can be configured to generate noise waveform, or triangle waveform. The output can be buffered for low impedance loads. The DAC can be driven manually or using the DMA and a timer. We will be developing projects in this Chapter using both methods.

### 9.2 Project 1: Sawtooth Waveform Generator with Manual DAC Driving

#### Description

In this project we will write a program to create a sawtooth waveform with 10 steps, having a period of 10 ms.

#### The aim

The aim of this project is to show how the DAC of the Nucleo-L476RG can be employed in a project.

#### Circuit diagram

Pin PA4 (pin 32 on connector CN7) of the Nucleo-L476RG is connected to a digital oscilloscope to record the waveform. In this project the PCSGU250 Velleman oscilloscope/transient analyser is used.



### Program listing

The period of the waveform is required to be 10 ms, having 10 steps. The length of each step should therefore be 1 ms. The **HAL\_Delay** function of the HAL library is not very accurate. It is therefore decided to use 1ms timer interrupts in this program and to send data to the DAC inside the timer interrupt service routine.

As was shown in Chapter 6, for 1-ms timer interrupt, assuming a timer prescaler value of 3999, the auto-reload value should be:

$$A = \frac{f \times I}{P + 1} - 1$$

or,

$$A = \frac{80,000,000 \times 0.001}{3999 + 1} - 1 = 19$$

The steps are given below.

- Start the STM32CubeMX program.
- Select the Nucleo-L476RG board as before.
- Give the name **SAWTOOTH** to the project.
- Select pin **PA4** and set it to **DAC1\_OUT1** (Figure 9.1).

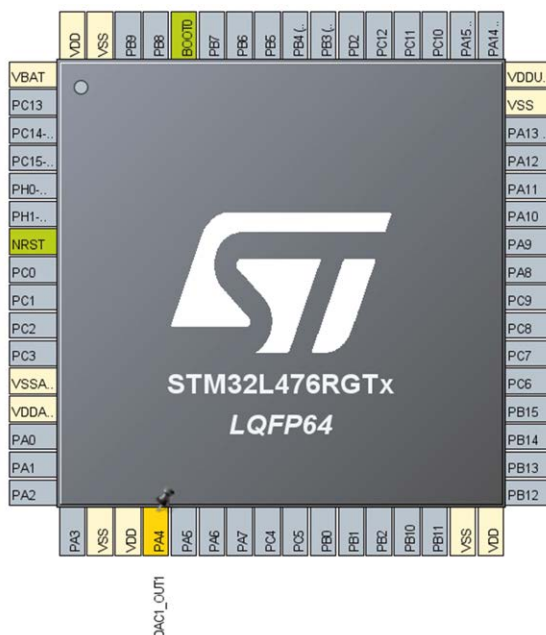


Figure 9.1: Setting PA4 to DAC1\_OUT1.

- Click **Clock Configuration** and set the clock to 80 MHz, making sure that there is clock to the system and to DAC1 (from bus **APB1**).

- Click **Analog** and then **DAC1**. Set **OUT1 mode** to **Connected to external pin and on-chip peripherals**
- In **Parameter Settings**, enable the **Output Buffer**.
- Click tab **Timers** and then click **TIM2** and set the **Clock Source** to **Internal Clock** (see Figure 9.2).

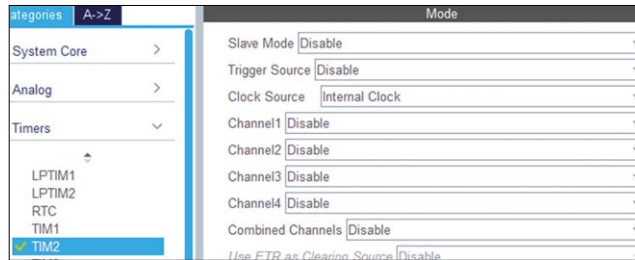


Figure 9.2: Setting the TIM2 clock.

- In **Parameter Settings**, set the **Prescaler** to **3999**, **Counter Mode** to **Down**, **Counter Period** to **19**, and enable **auto-reload preload** (Figure 9.3).

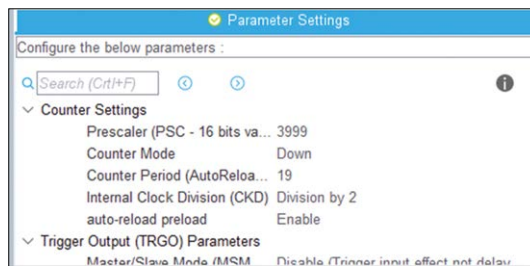


Figure 9.3: Configuring TIM2.

- Click **File**, followed by **Save** and click **YES** to generate code.
- Expand the project folder at the left pane. Click **Core**, followed by **Src** and then double click on **main.c** to display the main program.

Enter the DAC range as a floating-point number at the beginning of the program:

```
float DAC_RABGE = 4095;
```

Notice that the following statements are added to the program:

```
DAC_HandleTypeDef hdac1;
TIM_HandleTypeDef htim2;

void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_DAC1_Init(void);
static void MX_TIM2_Init(void);
```

Also, the following function is added:

```
static void MX_DAC1_Init(void)
{
 DAC_ChannelConfTypeDef sConfig = {0};

 hdac1.Instance = DAC1;
 if (HAL_DAC_Init(&hdac1) != HAL_OK)
 {
 Error_Handler();
 }

 sConfig.DAC_SampleAndHold = DAC_SAMPLEANDHOLD_DISABLE;
 sConfig.DAC_Trigger = DAC_TRIGGER_NONE;
 sConfig.DAC_OutputBuffer = DAC_OUTPUTBUFFER_ENABLE;
 sConfig.DAC_ConnectOnChipPeripheral = DAC_CHIPCONNECT_ENABLE;
 sConfig.DAC_UserTrimming = DAC_TRIMMING_FACTORY;
 if (HAL_DAC_ConfigChannel(&hdac1, &sConfig, DAC_CHANNEL_1) != HAL_OK)
 {
 Error_Handler();
 }
}
```

The timer interrupt service routine is called **HAL\_TIM\_PeriodElapsedCallback** and this function is called automatically every millisecond. Inside this function, a step of the saw-tooth waveform is sent to the DAC as in the following code:

```
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
 DACData = ((uint32_t)(j * DAC_RANGE));
 HAL_DAC_SetValue(&hdac1, DAC_CHANNEL_1, DAC_ALIGN_12B_R, DACData);
 j = j + 0.1;
 if(j > 1.0)j = 0.0;
}
```

The timer and the DAC are started as in the following statements:

```
HAL_TIM_Base_Start_IT(&htim2); // Start the timer
HAL_DAC_Start(&hdac1, DAC_CHANNEL_1); // Start the DAC
```

The DAC data format can be one of the following. In this program, 12-bit data is selected with the data aligned to the right:

- 8 bit right alignment using DAC\_ALIGN\_8B\_R
- 12 bit left alignment using DAC\_ALIGN\_12B\_L
- 12 bit right alignment using DAC\_ALIGN\_12B\_R

Now compile the program in **Release** mode and make sure there are no errors. Drag and drop the binary file **SAWTOOTH.bin** to device NUCLEO\_L476RG. Figure 9.4 shows the generated waveform on the oscilloscope. Here, the vertical axis was 1 V/division, and the horizontal axis was 5 ms/division.

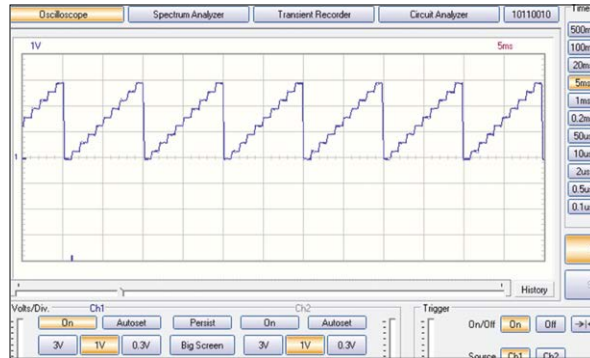


Figure 9.4: Generated waveform.

Figure 9.5 shows the complete program listing (comments have been removed for clarity).

```
/* USER CODE BEGIN Header */
/**
 * *****
 * @file : main.c
 * @brief : Main program body
 * *****
 * @attention
 *
 * <h2><center>© Copyright (c) 2020 STMicroelectronics.
 * All rights reserved.</center></h2>
 *
 * This software component is licensed by ST under BSD 3-Clause license,
 * the «License»; You may not use this file except in compliance with the
 * License. You may obtain a copy of the License at:
 *
 * opensource.org/licenses/BSD-3-Clause
 *
 * *****
 */
#include «main.h»
float DAC_RANGE = 4095;

DAC_HandleTypeDef hdac1;

TIM_HandleTypeDef htim2;

void SystemClock_Config(void);
```

```
static void MX_GPIO_Init(void);
static void MX_DAC1_Init(void);
static void MX_TIM2_Init(void);

float j = 0.0;
uint32_t DACData;

//
// This is the timer interrupt service routine. The program jumps here
// every ms. Here, data is sent to the DAC in 12-bit mode, right aligned
//
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
 DACData = ((uint32_t)(j * DAC_RANGE));
 HAL_DAC_SetValue(&hdac1, DAC_CHANNEL_1, DAC_ALIGN_12B_R, DACData);
 j = j + 0.1;
 if(j > 1.0)j = 0.0;
}

//
// Start of main program
//
int main(void)
{
 HAL_Init();

 SystemClock_Config();

 MX_GPIO_Init();
 MX_DAC1_Init();
 MX_TIM2_Init();

 HAL_TIM_Base_Start_IT(&htim2); // Start the timer
 HAL_DAC_Start(&hdac1, DAC_CHANNEL_1); // Start the DAC

 while (1)
 {
 }
}

void SystemClock_Config(void)
{
 RCC_OscInitTypeDef RCC_OscInitStruct = {0};
 RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

 RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
```

```
RCC_OscInitStruct.HSISState = RCC_HSI_ON;
RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
RCC_OscInitStruct.PLL.PLLM = 2;
RCC_OscInitStruct.PLL.PLLN = 20;
RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV7;
RCC_OscInitStruct.PLL.PLLQ = RCC_PLLQ_DIV2;
RCC_OscInitStruct.PLL.PLLR = RCC_PLLR_DIV2;
if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
{
 Error_Handler();
}

RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
 |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_4) != HAL_OK)
{
 Error_Handler();
}

if (HAL_PWREx_ControlVoltageScaling(PWR_REGULATOR_VOLTAGE_SCALE1) != HAL_OK)
{
 Error_Handler();
}
}

static void MX_DAC1_Init(void)
{
 DAC_ChannelConfTypeDef sConfig = {0};

 hdac1.Instance = DAC1;
 if (HAL_DAC_Init(&hdac1) != HAL_OK)
 {
 Error_Handler();
 }

 sConfig.DAC_SampleAndHold = DAC_SAMPLEANDHOLD_DISABLE;
 sConfig.DAC_Trigger = DAC_TRIGGER_NONE;
 sConfig.DAC_OutputBuffer = DAC_OUTPUTBUFFER_ENABLE;
 sConfig.DAC_ConnectOnChipPeripheral = DAC_CHIPCONNECT_ENABLE;
```

```
sConfig.DAC_UserTrimming = DAC_TRIMMING_FACTORY;
if (HAL_DAC_ConfigChannel(&hdac1, &sConfig, DAC_CHANNEL_1) != HAL_OK)
{
 Error_Handler();
}
}

static void MX_TIM2_Init(void)
{
 TIM_ClockConfigTypeDef sClockSourceConfig = {0};
 TIM_MasterConfigTypeDef sMasterConfig = {0};

 htim2.Instance = TIM2;
 htim2.Init.Prescaler = 3999;
 htim2.Init.CounterMode = TIM_COUNTERMODE_DOWN;
 htim2.Init.Period = 19;
 htim2.Init.ClockDivision = TIM_CLOCKDIVISION_DIV2;
 htim2.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_ENABLE;
 if (HAL_TIM_Base_Init(&htim2) != HAL_OK)
 {
 Error_Handler();
 }
 sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
 if (HAL_TIM_ConfigClockSource(&htim2, &sClockSourceConfig) != HAL_OK)
 {
 Error_Handler();
 }
 sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
 sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
 if (HAL_TIMEx_MasterConfigSynchronization(&htim2, &sMasterConfig) != HAL_OK)
 {
 Error_Handler();
 }
}

static void MX_GPIO_Init(void)
{
 /* GPIO Ports Clock Enable */
 __HAL_RCC_GPIOA_CLK_ENABLE();
}

void Error_Handler(void)
{
}
}
```

```

#ifdef USE_FULL_ASSERT

void assert_failed(uint8_t *file, uint32_t line)
{

}

#endif

/***** (C) COPYRIGHT STMicroelectronics *****/

```

*Figure 9.5: the SAWTOOTH program.*

### 9.3 Project 2: Squarewave Generator with Manual DAC Driving

#### Description

In this project we will generate a squarewave signal with a frequency of 500 kHz, having 50% duty cycle, i.e. 1 ms ON time, 1 ms OFF time

#### Program listing

The program is very similar to the one given in Figure 9.5, but here, the DAC port is toggled every millisecond. Part of the program listing is shown in Figure 9.6, where only the code inside the timer interrupt service routine is changed.

```

void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
 if(j == 0)
 {
 j = 1;
 DACData = 0;
 HAL_DAC_SetValue(&hdac1, DAC_CHANNEL_1, DAC_ALIGN_12B_R, DACData);
 }
 else
 {
 j = 0;
 DACData = DAC_RANGE;
 HAL_DAC_SetValue(&hdac1, DAC_CHANNEL_1, DAC_ALIGN_12B_R, DACData);
 }
}

```

*Figure 9.6: Timer interrupt service routine.*

Figure 9.7 shows the generated waveform on the oscilloscope. Here, the vertical axis was 1 V/division, and the horizontal axis was 5 ms/division.





Figure 9.7: Generated waveform.

### 9.4 Project 3: Sinewave Generator with Manual DAC Driving

#### Description

In this project we will generate a sine wave signal with the frequency of 5 Hz, i.e. a period of 200 ms. 200 points will be used in a period of the signal with each point having a length of 1 ms.

#### Program listing

The program is very similar to the one given in Figure 9.5, but here, the sinewave is sent to the DAC port. Part of the program listing is shown below, where only the code inside the timer interrupt service routine is changed. The program loop contains the following statements, which is repeated 200 times for each period:

```
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
 k = 0.5 + 0.5*sin(j*3.14159);
 p = (uint32_t)(k * DAC_RANGE);
 HAL_DAC_SetValue(&hdac1, DAC_CHANNEL_1, DAC_ALIGN_12B_R, p);
 j = j + 0.01;
 if(j > 2.0)j = 0.0;
}
```

The sinewave has positive and negative parts. In order to avoid having negative parts, the waveform is shifted up by 0.5 which corresponds to  $+3.3 \text{ V} \times 0.5 = +1.65 \text{ V}$  so that the wave goes from 0 to the peak voltage of +3.3 V.

Figure 9.8 shows the generated waveform on the oscilloscope. Here, the vertical axis was 1 V/division, and the horizontal axis was 50 ms/division.

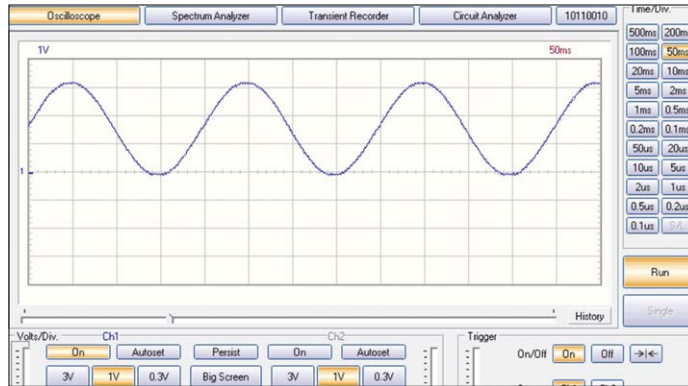


Figure 9.8: Generated waveform.

### 9.5 Project 4: Arbitrary Waveform Generator with Manual DAC Driving

#### Description

In this project we will generate an arbitrary waveform defined by the user. The generated waveform will have period of 20 ms, where the details of the waveform are as shown in Figure 9.9.

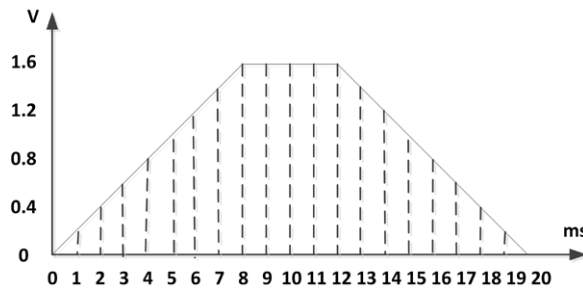


Figure 9.9: The waveform to be generated.

The waveform takes the following values:

| Time (ms) | Amplitude (V) | Time (ms) | Amplitude (V) |
|-----------|---------------|-----------|---------------|
| 0         | 0             | 11        | 1.6           |
| 1         | 0.2           | 12        | 1.6           |
| 2         | 0.4           | 13        | 1.4           |
| 3         | 0.6           | 14        | 1.2           |
| 4         | 0.8           | 15        | 1.0           |
| 5         | 1.0           | 16        | 0.8           |
| 6         | 1.2           | 17        | 0.6           |
| 7         | 1.4           | 18        | 0.4           |
| 8         | 1.6           | 19        | 0.2           |
| 9         | 1.6           | 20        | 0.0           |
| 10        | 1.6           |           |               |

## The aim

The aim of this project is to show how an arbitrary waveform can be generated.

## Program listing

The program is very similar to the one given in Figure 9.5, but here, the segments of the arbitrary waveform shown in Figure 9.9 are sent to the DAC port inside the timer interrupt. Part of the program listing that is modified is shown in Figure 9.10. Notice that the waveform points are stored in an array called **Waveform**, indexed by variable **k**.

```
int k = 0;
uint32_t p;
float Waveform[] = {0.0,0.2,0.4,0.6,0.8,1.0,1.2,1.4,1.6,1.6,1.6,1.6,1.6,
 1.4,1.2,1.0,0.8,0.6,0.4,0.2,0.0};

//
// This is the timer interrupt service routine. The program jumps here
// every ms. Here, data is sent to the DAC in 12-bit mode, right aligned
//
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
 p = (uint32_t)(Waveform[k] * DAC_RANGE/3.3);
 HAL_DAC_SetValue(&hdac1, DAC_CHANNEL_1, DAC_ALIGN_12B_R, p);
 k++;
 if(k >= 20)k = 0;
}
```

Figure 9.10: The part of the code that has been modified.

Figure 9.11 shows the generated waveform. In this figure the vertical axis is 1 V/division and the horizontal axis 5 ms/division.

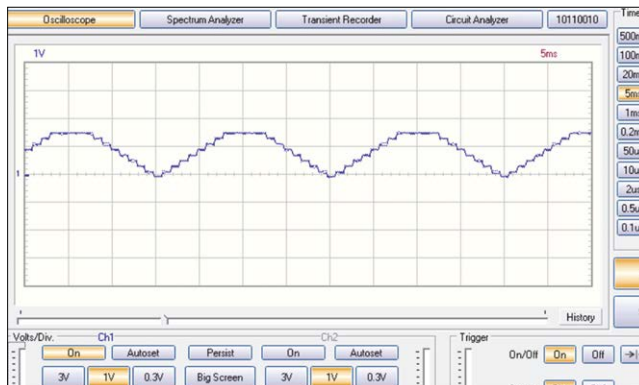


Figure 9.11: Generated waveform

## 9.6 Project 5: Arbitrary Waveform Generator with timer-based DMA

### Description

Perhaps the most efficient way to generate a waveform is to drive the DAC by using the DMA with a timer to trigger the conversion. Using DMA has the advantage that the DMA processing is done without the intervention of the CPU. In interrupt based manual modes (the case in the previous projects in this Chapter) the CPU must service the timer interrupts periodically which requires a lot of CPU intervention, especially when high rate of sampling is required. The DMA can be configured to operate in Circular mode so that the converted data is taken from the DAC and stored in a circular buffer without the intervention of the CPU. Figure 9.12 shows the concept of using the DMA with a timer to generate a waveform.

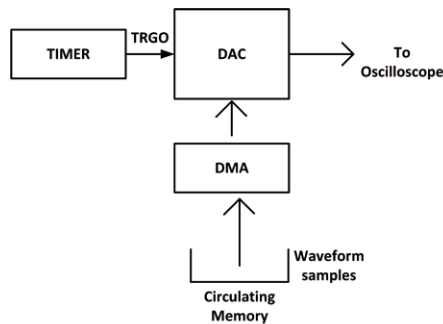


Figure 9.12: Using DMA.

The HAL function **HAL\_DAC\_Start\_DMA** is used to operate the DAC in DMA mode. This function has the following format:

**HAL\_DAC\_Start\_DMA(handle, Channel, Data, Length, Alignment)**

Where, **Channel** is the DAC channel used (DAC\_CHANNEL\_1 or DAC\_CHANNEL\_2), **Data** is the array of values to transfer in DMA mode with the specified **Length**, and **Alignment** is the output values in memory (DAC\_ALIGN\_8B\_R for 8-bit mode, DAC\_ALIGN\_12B\_L or DAC\_ALIGN\_12B\_R for 12-bit mode).

In this project we will generate the arbitrary waveform described in the previous project in Figure 9.9 using DMA with a timer. In this project we will use a timer configured to trigger the TRGO line at the required sampling intervals of 1 ms.

### Program listing

The steps are given below.

- Start the STM32CubeMX program.
- Select the Nucleo-L476RG board as before.
- Give the name **DACDMA** to the project.
- Select pin **PA4** and set it to **DAC1\_OUT1**.
- Click **Clock Configuration** and set the clock to 80 MHz, make sure that there is clock to the system and to DAC1 (from bus **APB1**).

- Click **Analog** and then **DAC1**. Set **OUT1 mode** to **Connected to external pin only**.
- In **Parameter Settings**, enable the **Output Buffer**, set the **Trigger** to **Timer 2 Trigger Out event** (Figure 9.13).

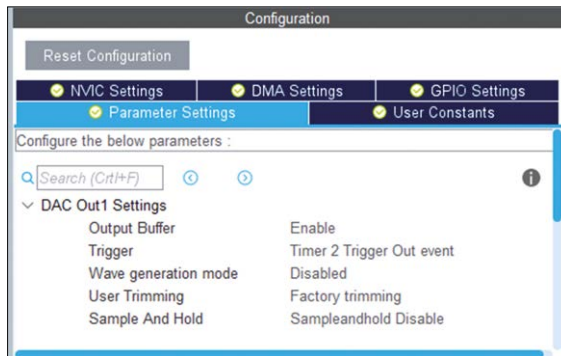


Figure 9.13: Parameter settings.

- Click **DMA Settings** under **Configuration** and click **Add** and select **DAC\_CH1**
- Click **DAC\_CH1** and set the **Mode** to **Circular**, **Data Width** to **Half Word** (Figure 9.14).

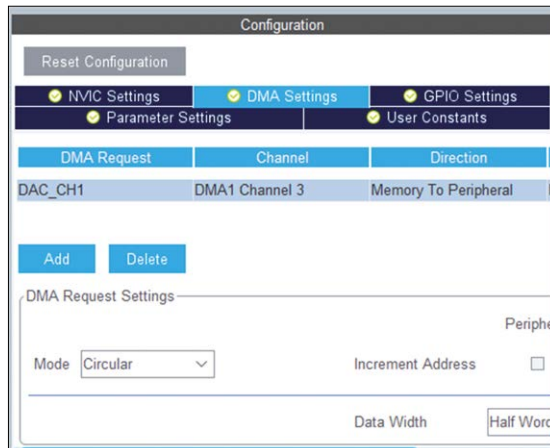


Figure 9.14: DMA settings.

- Click tab **Timers** and the click **TIM2** and set the **Clock Source** to **Internal Clock**
- In **Parameter Settings** select for 1ms interrupts. i.e. set the **Prescaler** to **3999**, **Counter Mode** to **Down**, **Counter Period** to **19**, and **Trigger Event Selection TRGO** to **Update Event** (Figure 9.15).

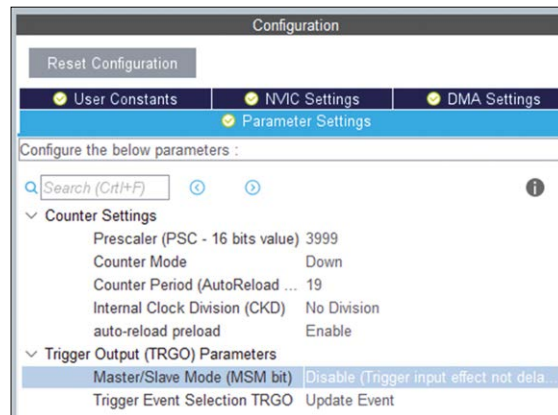


Figure 9.15: Parameter settings.

- Click **System Core** and then click **NVIC** and enable **DMA1 Channel3 global interrupt**.
- Click **File**, followed by **Save** and then click **YES** to generate code.
- Click **Core**, followed by **Src** and then double click **main.c** to open the main program.

At the beginning of the program define the number of samples and the DAC range:

```
#define Samples 21 // No of samples
float DAC_RANGE = 4095; // DAC range
```

Define the waveform samples in an array:

```
float Waveform[] = {0.0,0.2,0.4,0.6,0.8,1.0,1.2,1.4,1.6,1.6,1.6,1.6,1.6,
1.4,1.2,1.0,0.8,0.6,0.4,0.2,0.0};
```

Inside the main program copy the waveform samples to an **uint16\_t** type array called **DACValues**:

```
for(i = 0; i < 21; i++)DACValues[i] = (uint16_t)(Waveform[i]*DAC_RANGE / 3.3);
```

The next step is to initialize the DAC, start timer TIM2, and start the DMA. Notice that array **DACValues** is passed as an argument to the DMA start function with the length **Samples**:

```
HAL_DAC_Init(&hdac1);
HAL_TIM_Base_Start(&htim2);
HAL_DAC_Start_DMA(&hdac1, DAC_CHANNEL_1, (uint32_t*)DACValues, Samples,
DAC_ALIGN_12B_R);
```

Compile the program making sure there are no errors. Drag and drop the binary file **DACDMA.bin** to device NUCLEO\_L476RG.

Figure 9.16 shows the waveform displayed on the oscilloscope. In this display, the vertical

axis was 1 V/division and the horizontal axis, 5 ms/division.

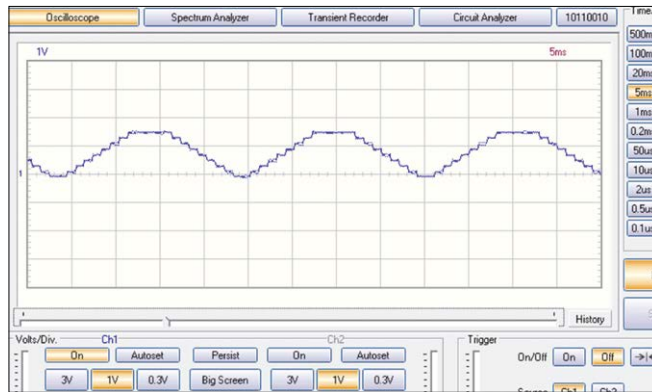


Figure 9.16: Waveform on the oscilloscope.

The program listing is shown in Figure 9.17 (comments have been removed).

```

/* USER CODE BEGIN Header */
/**
 * *****
 * @file : main.c
 * @brief : Main program body
 * *****
 * @attention
 *
 * <h2><center>© Copyright (c) 2020 STMicroelectronics.
 * All rights reserved.</center></h2>
 *
 * This software component is licensed by ST under BSD 3-Clause license,
 * the «License»; You may not use this file except in compliance with the
 * License. You may obtain a copy of the License at:
 *
 * opensource.org/licenses/BSD-3-Clause
 *
 * *****
 */
#include «main.h»

#define Samples 21 // No of samples
float DAC_RANGE = 4095; // DAC range

DAC_HandleTypeDef hdac1;
DMA_HandleTypeDef hdma_dac_ch1;

TIM_HandleTypeDef htim2;

```

```
void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_DMA_Init(void);
static void MX_DAC1_Init(void);
static void MX_TIM2_Init(void);

//
// Waveform samples
//
float Waveform[] = {0.0,0.2,0.4,0.6,0.8,1.0,1.2,1.4,1.6,1.6,1.6,1.6,
 1.4,1.2,1.0,0.8,0.6,0.4,0.2,0.0};

//
// Start of main program
//
int main(void)
{
 int i;
 uint16_t DACValues[21];

 HAL_Init();

 SystemClock_Config();

 MX_GPIO_Init();
 MX_DMA_Init();
 MX_DAC1_Init();
 MX_TIM2_Init();

 //
 // Copy the waveform samples to uint16_t array DACValues
 //
 for(i = 0; i < 21; i++)DACValues[i] = (uint16_t)(Waveform[i]*DAC_RANGE / 3.3);
 //
 // Initialize ADC, Start timer TIM2, start DAC DMA
 //
 HAL_DAC_Init(&hdac1);
 HAL_TIM_Base_Start(&htim2);
 HAL_DAC_Start_DMA(&hdac1, DAC_CHANNEL_1, (uint32_t*)DACValues, Samples, DAC_ALIGN_12B_R);

 while (1)
 {
 }
}
```



```
void SystemClock_Config(void)
{
 RCC_OscInitTypeDef RCC_OscInitStruct = {0};
 RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

 RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
 RCC_OscInitStruct.HSIState = RCC_HSI_ON;
 RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
 RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
 RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
 RCC_OscInitStruct.PLL.PLLM = 2;
 RCC_OscInitStruct.PLL.PLLN = 20;
 RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV7;
 RCC_OscInitStruct.PLL.PLLQ = RCC_PLLQ_DIV2;
 RCC_OscInitStruct.PLL.PLLR = RCC_PLLR_DIV2;
 if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
 {
 Error_Handler();
 }

 RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
 |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
 RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
 RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
 RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
 RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

 if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_4) != HAL_OK)
 {
 Error_Handler();
 }

 if (HAL_PWREx_ControlVoltageScaling(PWR_REGULATOR_VOLTAGE_SCALE1) != HAL_OK)
 {
 Error_Handler();
 }
}

static void MX_DAC1_Init(void)
{
 DAC_ChannelConfTypeDef sConfig = {0};

 hdac1.Instance = DAC1;
 if (HAL_DAC_Init(&hdac1) != HAL_OK)
 {

```

```

 Error_Handler();
}

sConfig.DAC_SampleAndHold = DAC_SAMPLEANDHOLD_DISABLE;
sConfig.DAC_Trigger = DAC_TRIGGER_T2_TRGO;
sConfig.DAC_OutputBuffer = DAC_OUTPUTBUFFER_ENABLE;
sConfig.DAC_ConnectOnChipPeripheral = DAC_CHIPCONNECT_DISABLE;
sConfig.DAC_UserTrimming = DAC_TRIMMING_FACTORY;
if (HAL_DAC_ConfigChannel(&hdac1, &sConfig, DAC_CHANNEL_1) != HAL_OK)
{
 Error_Handler();
}
}

static void MX_TIM2_Init(void)
{
 TIM_ClockConfigTypeDef sClockSourceConfig = {0};
 TIM_MasterConfigTypeDef sMasterConfig = {0};

 htim2.Instance = TIM2;
 htim2.Init.Prescaler = 3999;
 htim2.Init.CounterMode = TIM_COUNTERMODE_DOWN;
 htim2.Init.Period = 19;
 htim2.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
 htim2.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_ENABLE;
 if (HAL_TIM_Base_Init(&htim2) != HAL_OK)
 {
 Error_Handler();
 }
 sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
 if (HAL_TIM_ConfigClockSource(&htim2, &sClockSourceConfig) != HAL_OK)
 {
 Error_Handler();
 }
 sMasterConfig.MasterOutputTrigger = TIM_TRGO_UPDATE;
 sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
 if (HAL_TIMEx_MasterConfigSynchronization(&htim2, &sMasterConfig) != HAL_OK)
 {
 Error_Handler();
 }
}

static void MX_DMA_Init(void)
{

```

```
/* DMA controller clock enable */
__HAL_RCC_DMA1_CLK_ENABLE();

/* DMA interrupt init */
/* DMA1_Channel3_IRQn interrupt configuration */
HAL_NVIC_SetPriority(DMA1_Channel3_IRQn, 0, 0);
HAL_NVIC_EnableIRQ(DMA1_Channel3_IRQn);

}

/**
 * @brief GPIO Initialization Function
 * @param None
 * @retval None
 */
static void MX_GPIO_Init(void)
{

 /* GPIO Ports Clock Enable */
 __HAL_RCC_GPIOA_CLK_ENABLE();

}

void Error_Handler(void)
{

}

#ifdef USE_FULL_ASSERT

void assert_failed(uint8_t *file, uint32_t line)
{

}

#endif

/***** (C) COPYRIGHT STMicroelectronics *****END OF FILE*****/
```

*Figure 9.17. The DACDMA program.*

## 9.7 Hardware waveform generation

The STM32 is capable of generating triangular waves in hardware. The parameters of the generated triangular wave are shown in Figure 9.18.

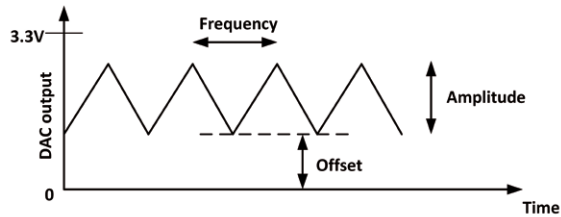


Figure 9.18: Generated triangular wave.

**Amplitude:** this is the peak to peak amplitude of the waveform, ranging from 0 to 0xFFFF, i.e. from 0 V to +3.3 V. The amplitude is directly connected to the offset value.

**Offset:** this is the minimum output value of the waveform. The sum of the offset and the amplitude cannot be greater than +3.3 V.

**Frequency:** this is the frequency of the waveform. The frequency is determined by the update frequency of the timer used and is given by:

$$f_{\text{update}} = 4 \times \text{amplitude} \times f_{\text{wave}}$$

if the timer is running at a frequency  $f_t$ , the required prescaler value is:

$$\text{prescaler} = f_t / f_{\text{update}}$$

For example, to generate a 25-Hz triangular wave with an amplitude equal to 2047 (1.65 V) with no offset, the prescaler of a timer running at 80 MHz needs to be configured to:

$$\begin{aligned} f_{\text{update}} &= 4 \times 2047 \times 25 = 204,700 \\ \text{prescaler} &= 80,000,000 / 204,700 = 390 \end{aligned}$$

The following function is used to generate triangular waveform, whose arguments are the handle, DAC channel number, and the required waveform amplitude:

**HAL\_DACEx\_TriangleWaveGenerate(handle, Channel , Amplitude);**

The procedure is as follows:

- Configure the DAC channel to be used (e.g. Channel 1, Pin PA4).
- Configure the timer associated with the DAC (e.g. TIM2).
- Configure the timer prescaler as calculated using the above formula (set the Counter period value to 1).
- Start the DAC using function **HAL\_DAC\_Start**.
- Configure the required offset value using function **HAL\_DAC\_SetValue**.
- Call function **HAL\_DACEx\_TriangleWaveGenerate** to generate the waveform (this is called automatically by the STM32CubeIDE software).

An example is given below to show how a triangular waveform can be generated in hardware.

## 9.8 Project 6: Hardware-based Triangular Waveform Generation

### Description

In this project we will generate a hardware based triangular waveform with the following specifications:

|                            |                        |
|----------------------------|------------------------|
| Frequency of the waveform: | 25 Hz (period = 40 ms) |
| Amplitude of the waveform: | +1.65 V (i.e. 2047)    |
| Offset:                    | 1 V (i.e. 1247)        |
| Timer clock frequency:     | 80 MHz                 |
| Timer prescaler value:     | 390                    |
| Counter period value:      | 1                      |
| Timer used:                | TIM2                   |
| DAC used:                  | Channel 1 (Pin PA4)    |

### Program listing

- Start the STM32CubeMX program.
- Select the Nucleo-L476RG board as before.
- Give the name **TRIANGLE** to the project.
- Select pin **PA4** and set it to **DAC1\_OUT1**.
- Click **Clock Configuration** and set the clock to 80 MHz, make sure that there is clock to the system and to DAC1 (from bus **APB1**).
- Click **Analog** and then **DAC1**. Set **OUT1 mode** to **Connected to external pin and on chip-peripherals**.
- In **Parameter Settings**, enable the **Output Buffer**, set **Trigger** to **Timer 2 Trigger Out event**, set **Wave generation mode** to **Triangle wave generation**, set the **Maximum Triangle Amplitude** to **2047** (1.65 V), see Figure 9.19.

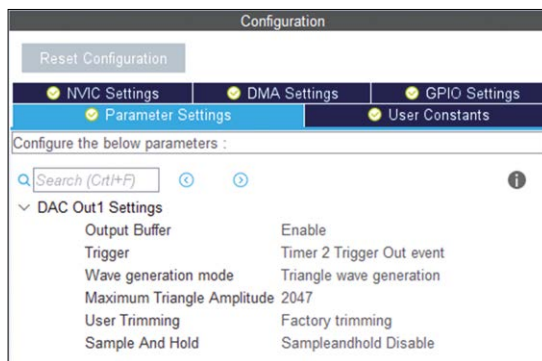


Figure 9.19: DAC parameter Settings.

- Click tab **Timers** and the click **TIM2** and set the **Clock Source** to **Internal Clock**.

- In **Parameter Settings** set the **Prescaler** to **390**, **Counter Mode** to **Down**, **Counter Period** to **1**, enable **auto-reload preload**, set Trigger Event Selection TRGO to Update Event (see Figure 9.20).

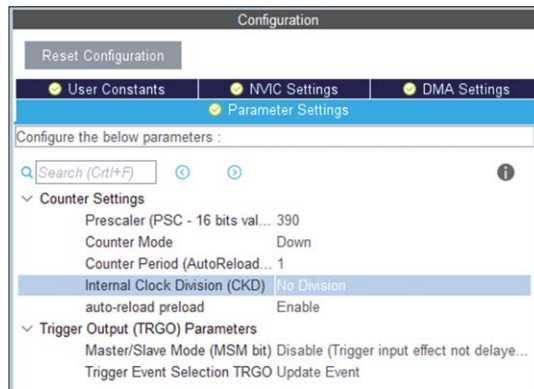


Figure 9.20: Timer Parameter Settings.

- Click **File**, followed by **Save** and click **YES** to generate code.

Define the Offset value at the beginning of the main program:

```
uint32_t Offset = 1247;
```

Start the timer, start the DAC, and set the Offset:

```
HAL_TIM_Base_Start_IT(&htim2);
HAL_DAC_Start(&hdac1, DAC_CHANNEL_1);
HAL_DAC_SetValue(&hdac1, DAC_CHANNEL_1, DAC_ALIGN_12B_R, Offset);
```

Compile the program and drag and drop the binary file **TRIANGLE.bin** to device NUCLEO\_L476RG. Connect the oscilloscope to pin PA4. The generated triangle waveform is shown in Figure 9.21. In this figure the vertical axis was 1 V/division, and the horizontal axis, 20 ms/division. Clearly, as expected the offset is 1 V and the amplitude is 1.65 V.

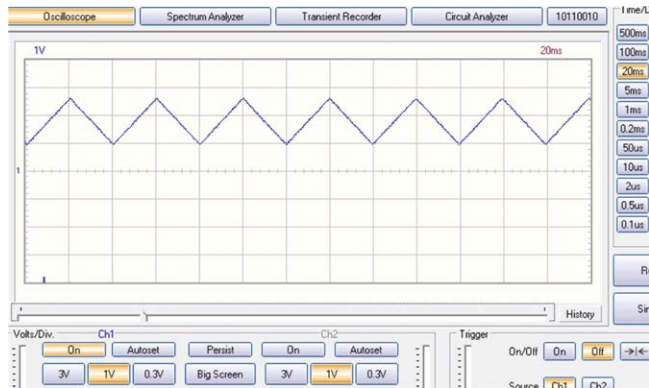


Figure 9.21: Generated triangle waveform.

Figure 9.22 shows the program listing (comments have been removed for clarity).

```

/* USER CODE BEGIN Header */
/**
 * *****
 * @file : main.c
 * @brief : Main program body
 * *****
 * @attention
 *
 * <h2><center>© Copyright (c) 2020 STMicroelectronics.
 * All rights reserved.</center></h2>
 *
 * This software component is licensed by ST under BSD 3-Clause license,
 * the «License»; You may not use this file except in compliance with the
 * License. You may obtain a copy of the License at:
 *
 * opensource.org/licenses/BSD-3-Clause
 *
 * *****
 */
#include «main.h»

```

```

DAC_HandleTypeDef hdac1;
TIM_HandleTypeDef htim2;

```

```

void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_DAC1_Init(void);
static void MX_TIM2_Init(void);

```

```

int main(void)
{

```

```

uint32_t Offset = 1247; // Set Offset to 1V
HAL_Init();

SystemClock_Config();

MX_GPIO_Init();
MX_DAC1_Init();
MX_TIM2_Init();

HAL_TIM_Base_Start_IT(&htim2); // Start the timer
HAL_DAC_Start(&hdac1, DAC_CHANNEL_1); // Start the DAC
HAL_DAC_SetValue(&hdac1, DAC_CHANNEL_1, DAC_ALIGN_12B_R, Offset); // Set Offset

while (1)
{
}
}

void SystemClock_Config(void)
{
 RCC_OscInitTypeDef RCC_OscInitStruct = {0};
 RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

 RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
 RCC_OscInitStruct.HSISState = RCC_HSI_ON;
 RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
 RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
 RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
 RCC_OscInitStruct.PLL.PLLM = 2;
 RCC_OscInitStruct.PLL.PLLN = 20;
 RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV7;
 RCC_OscInitStruct.PLL.PLLQ = RCC_PLLQ_DIV2;
 RCC_OscInitStruct.PLL.PLLR = RCC_PLLR_DIV2;
 if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
 {
 Error_Handler();
 }

 RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
 |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
 RCC_ClkInitStruct.SYSClkSource = RCC_SYSCCLKSOURCE_PLLCLK;
 RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCCLK_DIV1;
 RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
 RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

 if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_4) != HAL_OK)

```



```
{
 Error_Handler();
}

if (HAL_PWREx_ControlVoltageScaling(PWR_REGULATOR_VOLTAGE_SCALE1) != HAL_OK)
{
 Error_Handler();
}
}

static void MX_DAC1_Init(void)
{
 DAC_ChannelConfTypeDef sConfig = {0};

 hdac1.Instance = DAC1;
 if (HAL_DAC_Init(&hdac1) != HAL_OK)
 {
 Error_Handler();
 }

 sConfig.DAC_SampleAndHold = DAC_SAMPLEANDHOLD_DISABLE;
 sConfig.DAC_Trigger = DAC_TRIGGER_T2_TRGO;
 sConfig.DAC_OutputBuffer = DAC_OUTPUTBUFFER_ENABLE;
 sConfig.DAC_ConnectOnChipPeripheral = DAC_CHIPCONNECT_ENABLE;
 sConfig.DAC_UserTrimming = DAC_TRIMMING_FACTORY;
 if (HAL_DAC_ConfigChannel(&hdac1, &sConfig, DAC_CHANNEL_1) != HAL_OK)
 {
 Error_Handler();
 }

 if (HAL_DACEx_TriangleWaveGenerate(&hdac1, DAC_CHANNEL_1, DAC_TRIANGLEAMPLITUDE_2047) != HAL_OK)
 {
 Error_Handler();
 }
}

static void MX_TIM2_Init(void)
{
 TIM_ClockConfigTypeDef sClockSourceConfig = {0};
 TIM_MasterConfigTypeDef sMasterConfig = {0};

 htim2.Instance = TIM2;
 htim2.Init.Prescaler = 390;
 htim2.Init.CounterMode = TIM_COUNTERMODE_DOWN;
 htim2.Init.Period = 1;
```

```

htim2.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
htim2.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_ENABLE;
if (HAL_TIM_Base_Init(&htim2) != HAL_OK)
{
 Error_Handler();
}
sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
if (HAL_TIM_ConfigClockSource(&htim2, &sClockSourceConfig) != HAL_OK)
{
 Error_Handler();
}
sMasterConfig.MasterOutputTrigger = TIM_TRGO_UPDATE;
sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
if (HAL_TIMEx_MasterConfigSynchronization(&htim2, &sMasterConfig) != HAL_OK)
{
 Error_Handler();
}
}

static void MX_GPIO_Init(void)
{
 /* GPIO Ports Clock Enable */
 __HAL_RCC_GPIOA_CLK_ENABLE();
}

void Error_Handler(void)
{
}

#ifdef USE_FULL_ASSERT

void assert_failed(uint8_t *file, uint32_t line)
{
}

#endif

/***** (C) COPYRIGHT STMicroelectronics *****/

```

*Figure 9.22: The TRIANGLE program.*

### 9.9 Noise signal generation

Noise is useful in audio and RF applications. The STM32 MCU is perfectly able to generate noise waves using a pseudo-random generator as shown in Figure 9.23.

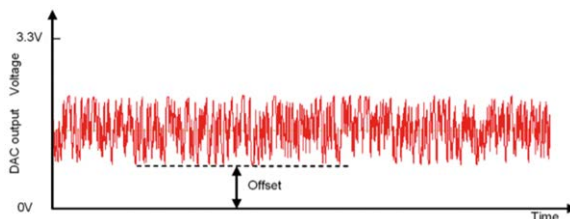


Figure 9.23: Noise waveform.

Further information on waveform generation can be obtained from the following STMicroelectronics Application Note:

*AN3126 Application Note Audio and waveform generation using the DAC in STM32 products*

The following HAL function is used to generate noise from the DAC. As with the triangle wave generation a timer is used to trigger the DAC:

**HAL\_DACEx\_NoiseWaveGenerate(handle, Channel, Amplitude);**

To program for noise generation, set the **Wave generation mode** in DAC **Parameter Settings** to **Noise wave generation**, and select the **Noise Amplitude** as required (see Figure 9.24).

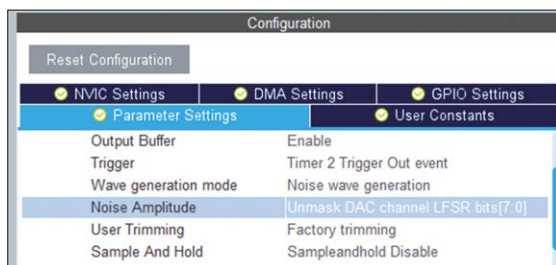


Figure 9.24: Configuring the DAC for noise generation.

## 9.10 Summary

In this Chapter we have learned how to program the DAC. Several working and fully tested projects are given in the Chapter. In the next Chapter we will be focusing on the pulse-width-Modulation (PWM) and learn how to program the Nucleo-L476RG development board to generate PWM waveforms.

## CHAPTER 10 • Pulsewidth Modulation (PWM)

### 10.1 Overview

Pulsewidth Modulation (PWM) is commonly used to drive heavy loads such as motors, actuators and so on. As we shall see in this Chapter, PWM is basically a positive squarewave signal whose pulsewidth can be changed. By changing the pulsewidth, we can effectively change the **average** voltage supplied to the load.

### 10.2 Basic theory of pulsewidth modulation

Pulsewidth Modulation (PWM) is a commonly used technique for controlling the power delivered to analogue loads using digital waveforms. Although analogue voltages (and currents) can be used to control the delivered power, they have several drawbacks. Controlling large analogue loads require large voltages and currents that cannot easily be obtained using standard analogue circuits and DACs. Precision analogue circuits can be heavy, large, and expensive and they are also sensitive to noise. By using the PWM technique the average value of voltage (and current) fed to a load is controlled by switching the supply voltage ON and OFF at a fast rate. The longer the power on time, the higher is the voltage supplied to the load.

Figure 10.1 shows a typical PWM waveform where the signal is basically a repetitive positive pulse, having the period  $t$ , ON time  $t_{ON}$ , and OFF time of  $(t - t_o)$  seconds. The minimum and maximum values of the voltage supplied to the load are 0 and  $V_p$  respectively. The PWM switching frequency is usually set to be very high (usually in the order of several kHz) so that it does not affect the load that uses the power. The main advantage of PWM is that the load is operated efficiently since the power loss in the switching device is very low. When the switch is ON there is practically no voltage drop across the switch, and when the switch is OFF there is no current supplied to the load.

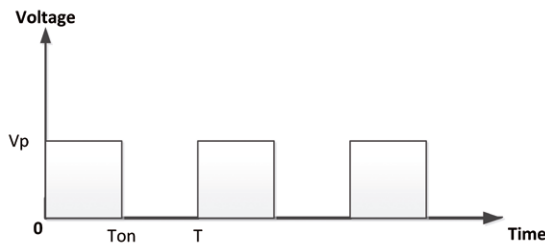


Figure 10.1: PWM waveform.

The duty cycle  $D$  of a PWM waveform is defined as the ratio of the ON time to its period. Expressed mathematically,

$$D = t_{ON} / t$$

The duty cycle is usually expressed as a percentage and therefore,

$$D = (t_{ON} / t_{OFF}) \times 100 \text{ [\%]}$$

By varying the duty cycle between 0% and 100% we can effectively control the average voltage supplied to the load between 0 and  $V_p$ .

The average value of the voltage applied to the load can be calculated by considering a general PWM waveform shown in Figure 10.1. The average value  $A$  of waveform  $f(t)$  with period  $T$  and peak value  $y_{\max}$  and minimum value  $y_{\min}$  is calculated as:

$$A = \frac{1}{T} \int_0^T f(t) dt$$

or,

$$A = \frac{1}{T} \left( \int_0^{T_{ON}} y_{\max} dt + \int_{T_{ON}}^T y_{\min} dt \right)$$

In a PWM waveform  $y_{\min} = 0$  and the above equation becomes

$$A = \frac{1}{T} (T_{ON} y_{\max})$$

or,  $A = D y_{\max}$

As it can be seen from the above equation, the average value of the voltage supplied to the load is directly proportional to the duty cycle of the PWM waveform and by varying the duty cycle we control the average load voltage. Figure 10.2 shows the average voltage for different values of the duty cycle.

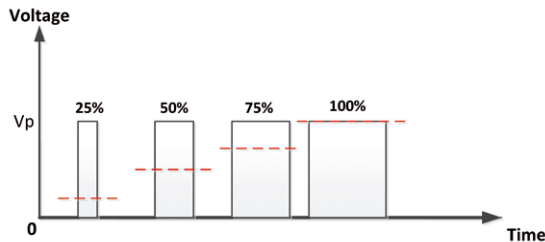


Figure 10.2: Average voltage (shown as dashed line) supplied to a load.

It is interesting to notice that with correct low-pass filtering, the PWM can be used as a DAC if the MCU does not have a DAC channel. By varying the duty cycle we can effectively vary the average analogue voltage supplied to the load.

### 10.3 Operation of the PWM

The timer module on the STM32 MCU can operate in several modes, one of which is the PWM mode. In this mode, the timer is clocked from an internal clock and counts up to the value of the auto-reload register (ARRx). Then, the output channel pin is driven HIGH. This pin remains HIGH until the timer count reaches the capture/compare (CCRx) register value, and at this time the output channel pin is driven LOW. The pin remains LOW until the coun-

ter counts up again and this process is repeated. The resulting waveform at the channel pin is the PWM signal. The frequency of the PWM signal is determined by the internal clock, the prescaler, and the auto-reload register ARR<sub>x</sub>. The duty cycle of the PWM is defined by the CCR<sub>x</sub> register value.

Figure 10.3 shows how a PWM waveform is generated. Notice in this figure that by changing the CCR<sub>x</sub> register value we can effectively change the duty cycle of the signal. For example, CCR1 is larger than CCR2 and the generated waveform has higher duty cycle (wider ON time than the OFF time). By changing the frequency of the internal clock, or the prescaler value, or the auto-reload register value we can change the frequency of the generated PWM waveform.

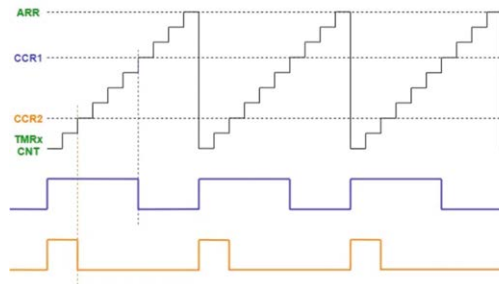


Figure 10.3: Generating a PWM waveform.

The STM32 microcontroller incorporates large number of timers. Most timers can have up to four PWM channels. The same timer can be used to generate another PWM signal having the same period, but a different duty cycle. Each channel has independent CCR<sub>x</sub> register, but shared ARR<sub>x</sub> register. It is therefore possible to generate a PWM signal for each channel of a timer where the period of the waveform is the same but different channels can have different duty cycles.

Some PWM pins of the STM32L476RG MCU are listed below (the first number is the timer number, the second number, the channel number). Notice that same timer/channel numbers may be available on different pins. For example, PWM 2/2 is available on pins PA1 and PB3:

|      |     |
|------|-----|
| PA8  | 1/1 |
| PA9  | 1/2 |
| PA10 | 1/3 |
| PA11 | 1/4 |
| PA15 | 2/1 |
| PA1  | 2/2 |
| PB10 | 2/3 |
| PB11 | 2/4 |
| PA6  | 3/1 |
| PA7  | 3/2 |
| PB0  | 3/3 |
| PB1  | 3/4 |

Generating PWM waveforms using the STM32CubeIDE is an easy task since most of the configuration can be done graphically and easily. A simple example project is given below.

### 10.4 Project 1: Mosquito Repeller

#### Description

It is a well-known fact that sound at 40 kHz ultrasonic frequencies can be used to repel mosquitos. In this project a 40-kHz PWM waveform is generated. This waveform can be used to drive a piezo transducer to generate ultrasonic sound and repel mosquitos.

#### The aim

The aim of this project is to show how a PWM waveform can be generated using the Nucleo-L476RG board.

#### Circuit diagram

Pin PB3 (PWM\_OUT2) of the Nucleo-L476RG board is connected to a piezo ultrasonic transducer (e.g. SQ-40-T-10B or UST-40T) through a type 2N4401 switching transistor. Figure 10.4 shows the circuit diagram of the project.

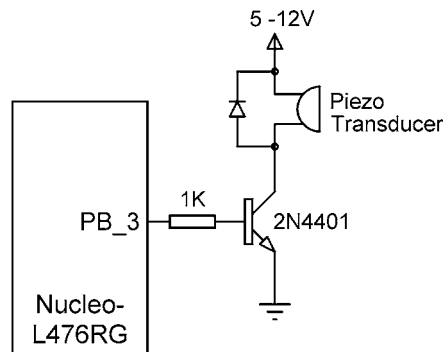


Figure 10.4: Circuit diagram of the project.

#### Program listing

The generation of a PWM waveform depends on the use of a timer. In the PWM mode when the counter value reaches a certain value it triggers and sets the PWM output high. The PWM output is cleared to 0 when the counter reaches to another value where at this point the counter is reset to 0. Because of this we have to set up a timer. In this project TIM2 is used.

The steps are given below. We shall be using PWM at pin PB3:

- Start the STM32CubeIDE program.
- Select the Nucleo-L476RG board as before.
- Name the project as **PWM**.
- Click on pin **PB3** and enable **TIM2\_CH2**.
- Click **TIM2** under **Timers** and select **Internal Clock** as the **Clock Source**, enable **PWM Generation CH2** as shown in Figure 10.5.

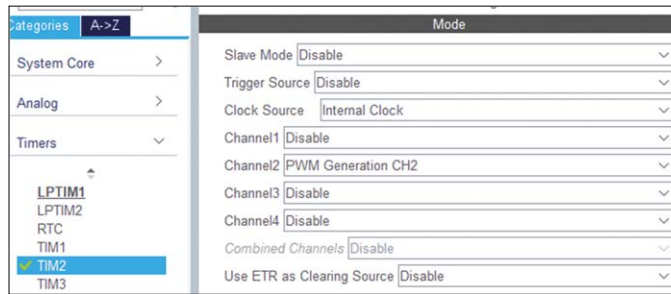


Figure 10.5: Enable PWM on channel 2 at TIM2.

- In **Parameter Settings**, set the **Counter Mode** to **Up**.
- We now have to set the timer parameters for the required PWM specifications. The required PWM period is 0.025 ms (25  $\mu$ s, 40 kHz) and the required PWM pulsewidth is 0.0125 ms (12.5  $\mu$ s, i.e. 50% duty cycle). The timer clock frequency is set to 80 MHz (period = 0.0125  $\mu$ s). If we set the prescaler to 80, then  $80 \times 0.0125 = 1 \mu$ s. Therefore, for a period of 25  $\mu$ s, the counter should be set to  $25 / 1 = 25$ . Enable **auto-reload preload**. Leave the other settings as they are (see Figure 10.6). The pulsewidth can be selected later when the code is generated.

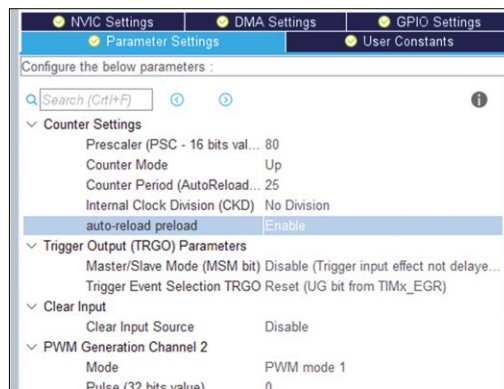


Figure 10.6: Parameter settings.

- Click **Clock Configuration** and make sure the system clock as well as the clock to the APB1 bus have been configured to 80 MHz.
- Click **File**, followed by **Save** and click **YES** to generate code
- Click **Core**, followed by **Src**, and double click **main.c** to open the main program

You will notice that the following lines are added to the main program:

```
TIM_HandleTypeDef htim2;
static void MX_TIM2_Init(void);
void HAL_TIM_MspPostInit(TIM_HandleTypeDef *htim);
MX_TIM2_Init();
```



In addition, the following function is added to the program:

```
static void MX_TIM2_Init(void)
{
 TIM_ClockConfigTypeDef sClockSourceConfig = {0};
 TIM_MasterConfigTypeDef sMasterConfig = {0};
 TIM_OC_InitTypeDef sConfigOC = {0};

 htim2.Instance = TIM2;
 htim2.Init.Prescaler = 80;
 htim2.Init.CounterMode = TIM_COUNTERMODE_UP;
 htim2.Init.Period = 25;
 htim2.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
 htim2.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_ENABLE;
 if (HAL_TIM_Base_Init(&htim2) != HAL_OK)
 {
 Error_Handler();
 }
 sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
 if (HAL_TIM_ConfigClockSource(&htim2, &sClockSourceConfig) != HAL_OK)
 {
 Error_Handler();
 }
 if (HAL_TIM_PWM_Init(&htim2) != HAL_OK)
 {
 Error_Handler();
 }
 sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
 sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
 if (HAL_TIMEx_MasterConfigSynchronization(&htim2, &sMasterConfig) != HAL_
OK)
 {
 Error_Handler();
 }
 sConfigOC.OCMode = TIM_OCMODE_PWM1;
 sConfigOC.Pulse = 0;
 sConfigOC.OCpolarity = TIM_OCPOLARITY_HIGH;
 sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
 if (HAL_TIM_PWM_ConfigChannel(&htim2, &sConfigOC, TIM_CHANNEL_2) != HAL_OK)
 {
 Error_Handler();
 }
 HAL_TIM_MspPostInit(&htim2);
}
```

Now, we need to set the pulsewidth to 12.5  $\mu\text{s}$  for 50% duty cycle. This is done in function **MX\_TIM2\_Init** above where the parameter **sConfigOC.Pulse = 0**; specifies the pulsewidth. In this project, set this parameter to 12.5.

Enter the following function before the program loop to start TIM2 in PWM mode:

```
HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_2);
```

Compile the program in **Release** mode and drag and drop the binary file **PWM.bin** to device NUCLEO\_L476RG. Figure 10.7 shows the generated waveform on the oscilloscope. Here, the vertical axis was 1 V/division, and the horizontal axis was 10  $\mu\text{s}$ /division.



Figure 10.7: Generated waveform.

Figure 10.8 shows the complete program listing (comments have been removed for clarity).

```
/* USER CODE BEGIN Header */
/**
 * *****
 * @file : main.c
 * @brief : Main program body
 * *****
 * @attention
 *
 * <h2><center>© Copyright (c) 2020 STMicroelectronics.
 * All rights reserved.</center></h2>
 *
 * This software component is licensed by ST under BSD 3-Clause license,
 * the «License»; You may not use this file except in compliance with the
 * License. You may obtain a copy of the License at:
 *
 * opensource.org/licenses/BSD-3-Clause
 *
 * *****
 */
#include «main.h»
```

```
TIM_HandleTypeDef htim2;

void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_TIM2_Init(void);

int main(void)
{
 HAL_Init();

 SystemClock_Config();

 MX_GPIO_Init();
 MX_TIM2_Init();
 HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_2);

 while (1)
 {
 }
}

void SystemClock_Config(void)
{
 RCC_OscInitTypeDef RCC_OscInitStruct = {0};
 RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

 RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
 RCC_OscInitStruct.HSISState = RCC_HSI_ON;
 RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
 RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
 RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
 RCC_OscInitStruct.PLL.PLLM = 2;
 RCC_OscInitStruct.PLL.PLLN = 20;
 RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV7;
 RCC_OscInitStruct.PLL.PLLQ = RCC_PLLQ_DIV2;
 RCC_OscInitStruct.PLL.PLLR = RCC_PLLR_DIV2;
 if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
 {
 Error_Handler();
 }

 RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
 |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
 RCC_ClkInitStruct.SYSClkSource = RCC_SYSCLKSOURCE_PLLCLK;
 RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
 RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
```

```

RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_4) != HAL_OK)
{
 Error_Handler();
}

if (HAL_PWREx_ControlVoltageScaling(PWR_REGULATOR_VOLTAGE_SCALE1) != HAL_OK)
{
 Error_Handler();
}
}

static void MX_TIM2_Init(void)
{
 TIM_ClockConfigTypeDef sClockSourceConfig = {0};
 TIM_MasterConfigTypeDef sMasterConfig = {0};
 TIM_OC_InitTypeDef sConfigOC = {0};

 htim2.Instance = TIM2;
 htim2.Init.Prescaler = 80;
 htim2.Init.CounterMode = TIM_COUNTERMODE_UP;
 htim2.Init.Period = 25;
 htim2.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
 htim2.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_ENABLE;
 if (HAL_TIM_Base_Init(&htim2) != HAL_OK)
 {
 Error_Handler();
 }
 sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
 if (HAL_TIM_ConfigClockSource(&htim2, &sClockSourceConfig) != HAL_OK)
 {
 Error_Handler();
 }
 if (HAL_TIM_PWM_Init(&htim2) != HAL_OK)
 {
 Error_Handler();
 }
 sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
 sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
 if (HAL_TIMEx_MasterConfigSynchronization(&htim2, &sMasterConfig) != HAL_OK)
 {
 Error_Handler();
 }
 sConfigOC.OCMode = TIM_OCMODE_PWM1;
 sConfigOC.Pulse = 12.5;

```

```
sConfigOC.OCpolarity = TIM_OCPOLARITY_HIGH;
sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
if (HAL_TIM_PWM_ConfigChannel(&htim2, &sConfigOC, TIM_CHANNEL_2) != HAL_OK)
{
 Error_Handler();
}
HAL_TIM_MspPostInit(&htim2);
}

static void MX_GPIO_Init(void)
{
 /* GPIO Ports Clock Enable */
 __HAL_RCC_GPIOB_CLK_ENABLE();
}

void Error_Handler(void)
{
}

#ifdef USE_FULL_ASSERT

void assert_failed(uint8_t *file, uint32_t line)
{
}

#endif

/***** (C) COPYRIGHT STMicroelectronics *****END OF FILE*****/
```

*Figure 10.8: The PWM program.*

### Changing the duty cycle

In the program in Figure 10.8, the duty cycle is fixed at 50%. HAL library provides functions for reading or modifying the duty cycle. The following function reads the current duty cycle into variable called **dutyCycle**:

```
uint16_t dutyCycle = HAL_TIM_ReadCapturedValue(&htim2, TIM_CHANNEL_2);
```

To read the current value of the auto-reload register call function:

```
__HAL_TIM_GET_AUTORELOAD(&htim2))
```

To read the current value of the auto-reload register value, call the following function:

```
__HAL_TIM_GET_AUTORELOAD(&htim2))
```

To change the duty cycle, we have to change the timer capture/compare register value. For example, to set the duty cycle to 25% call the following function (you can set the **sConfigOC.Pulse** value back to 0) before the program loop and set the duty cycle parameter (last parameter) to 6.25. Notice that the duty cycle is a percentage of the auto-reload value. Since the auto-reload value is 50µs, for 25% duty cycle we must set the duty cycle to 6.25 µs:

```
__HAL_TIM_SET_COMPARE(&htim2, TIM_CHANNEL_2, 6.25);
```

We can also read the value of the auto-reload register and divide it by 4 and use the result in the above function in order to set the duty cycle to 25%.

The program code to set the duty cycle to 25% in the main program is shown below:

```
int main(void)
{
 HAL_Init();
 SystemClock_Config();

 MX_GPIO_Init();
 MX_TIM2_Init();
 HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_2);
 __HAL_TIM_SET_COMPARE(&htim2, TIM_CHANNEL_2, 6.25);

 while (1)
 {
 }
}
```

We could also change the duty cycle as a percentage using the following code. Here, **ReqDutyCycle** is the required duty cycle (e.g. 25%):

```
ReqDutyCycle = 25.0;
AutoReload = __HAL_TIM_GET_AUTORELOAD(&htim2);
SetDutyCycle = AutoReload * ReqDutyCycle / 100.0;
__HAL_TIM_SET_COMPARE(&htim2, TIM_CHANNEL_2, SetDutyCycle);
```

The above code can be created as a function as shown below.

```
void DutyCycle(float Duty)
{
 uint16_t AutoReload, SetDutyCycle;
 AutoReload = __HAL_TIM_GET_AUTORELOAD(&htim2);
 SetDutyCycle = AutoReload * Duty / 100.0;
 __HAL_TIM_SET_COMPARE(&htim2, TIM_CHANNEL_2, SetDutyCycle);
}
```

Call the function from the main program. e.g. **DutyCycle(25.0)**

The program code except the comments, clock code, and the timer code (which are same as in the previous project) to set the duty cycle to 25% is shown in Figure 10.9.

```
#include «main.h»

TIM_HandleTypeDef htim2;

void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_TIM2_Init(void);

void DutyCycle(float Duty)
{
 uint16_t AutoReload, SetDutyCycle;
 AutoReload = __HAL_TIM_GET_AUTORELOAD(&htim2);
 SetDutyCycle = AutoReload * Duty / 100.0;
 __HAL_TIM_SET_COMPARE(&htim2, TIM_CHANNEL_2, SetDutyCycle);
}

int main(void)
{
 HAL_Init();

 SystemClock_Config();

 MX_GPIO_Init();
 MX_TIM2_Init();
 HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_2);

 DutyCycle(25.0);

 while (1)
 {
 }
}
```

*Figure 10.9: Using a function to change the duty cycle.*

## 10.5 Project 2: Continuously Variable Duty Cycle

### Description

In this project a 40-kHz PWM waveform is generated as in the previous project. Here, the duty cycle is incremented every 100 ms up to 100%, and then decremented every 100 ms to 0%. This process is repeated.

**The aim**

The aim of this project is to show how the duty cycle of a PWM waveform can be increased and decreased every second.

**Circuit diagram**

The circuit diagram of the project is same as in Figure 10.4, where pin PB3 is used as the PWM output.

**Program listing**

The program is very similar to the previous one, except that the duty cycle is changed continuously inside the program loop. Figure 10.10 shows the program listing (the comments, clock functions, and timer functions are removed as they are same as in the previous project).

```
#include «main.h»

TIM_HandleTypeDef htim2;

void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_TIM2_Init(void);

void DutyCycle(float Duty)
{
 uint16_t AutoReload, SetDutyCycle;
 AutoReload = __HAL_TIM_GET_AUTORELOAD(&htim2);
 SetDutyCycle = AutoReload * Duty / 100.0;
 __HAL_TIM_SET_COMPARE(&htim2, TIM_CHANNEL_2, SetDutyCycle);
}

int main(void)
{
 int i;
 HAL_Init();

 SystemClock_Config();

 MX_GPIO_Init();
 MX_TIM2_Init();
 HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_2);

 while (1)
 {
 for(i = 0; i <= 100; i++)
 {
```



```
 DutyCycle(i);
 HAL_Delay(100);
 }

 for(i = 100; i >= 0; i--)
 {
 DutyCycle(i);
 HAL_Delay(100);
 }
}
```

*Figure 10.10: Program listing.*

## 10.6 Project 3: Multiple PWM Waveforms

### Description

There are applications where we may want to generate multiple PWM waveforms. If the frequency of the required waveforms is to be the same, we can use the channels of the same timer as up to 4 channels of a given timer can be used to generate PWM waveforms. If on the other hand the frequencies are to be different then we should use different timers. In this project a 40 kHz PWM waveform is generated from 3 channels of timer 2. Although the frequencies of these waveforms will all be 40 kHz, their duty cycles can all be different if required.

The following GPIO pins are used as PWM outputs in this project:

|      |                          |
|------|--------------------------|
| PA15 | Timer 2, Channel 1 (2/1) |
| PB3  | Timer 2, Channel 2 (2/2) |
| PB10 | Timer 2, Channel 3 (2/3) |

### The aim

The aim of this project is to show how multiple PWM waveforms can be generated with different duty cycles.

### Program listing

- Start the STM32CubeIDE program.
- Select the Nucleo-L476RG board as before.
- Name the project as **PWM3**.
- Click on pin **PA15** and enable **TIM2\_CH1**. Also, click on pin **PB3** and enable **TIM2\_CH2**, and click pin **PB10** and enable **TIM2\_CH3** (Figure 10.11).



Figure 10.11: Selecting the Timer pins.

- Click **TIM2** under **Timers** and select **Internal Clock** as the **Clock Source**, enable **PWM Generation CH1**, **PWM Generation CH2**, and **PWM Generation CH3** as shown in Figure 10.12.

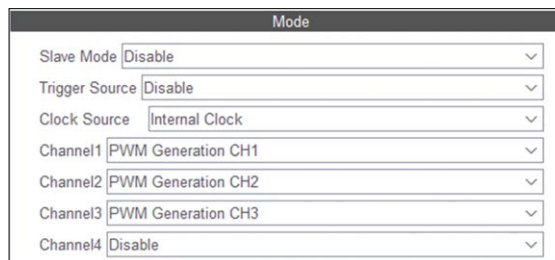


Figure 10.12: Enabling PWM on channels 1, 2, 3, on TIM2.

- In **Parameter Settings**, set the **Counter Mode** to **Up**.
- We now have to set the timer parameters for the required PWM specifications. The required PWM period is 0.025 ms (25  $\mu$ s, 40 kHz) and the required pulsewidth is 0.0125 ms (12.5  $\mu$ s, i.e. 50% duty cycle). The timer clock frequency is set to 80 MHz (period = 0.0125  $\mu$ s). If we set the prescaler to 80, then  $80 \times 0.0125 = 1 \mu$ s. Therefore, for a period of 25  $\mu$ s, the counter should be set to  $25 / 1 = 25$ . Enable **auto-reload preload**. Leave the other settings as they are (see Figure 10.13). The pulsewidth can be selected later when the code is generated.

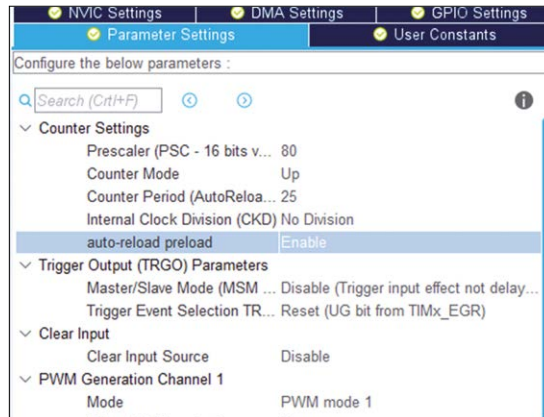


Figure 10.13: Parameter settings.

- Click **Clock Configuration** and make sure the system clock and also clock to APB1 bus have been configured to 80 MHz.
- Click **File**, followed by **Save** and click **YES** to generate code.
- Click **Core**, followed by **Src**, followed and double click **main.c** to open the main program.
- Create a new function called **DutyCycle** with two parameters: the required duty cycle as a percentage, and the channel number. The duty cycles are set as follows:

Channel 1: 6.25 (25%)  
 Channel 2: 12.50 (50%)  
 Channel 3: 18.75 (75%)

- Enter the following function before the program loop to start TIM2 in PWM mode:

```
HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_2);
```

- Compile the program in **Release** mode and drag and drop the binary file **PWM.bin** to device NUCLEO\_L476RG.
- Function **DutyCycle** sets the duty cycle for a channel of timer TIM2, like this:

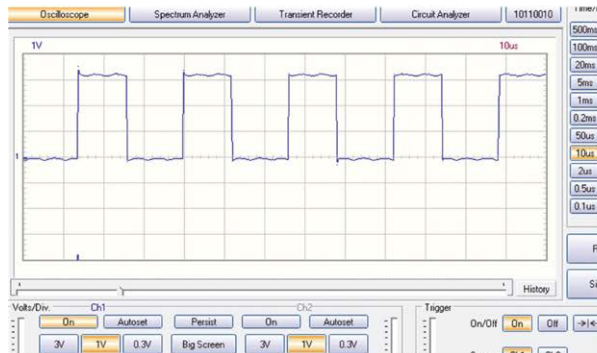
```
void DutyCycle(float Duty, int Channel)
{
 uint16_t AutoReload, SetDutyCycle;
 AutoReload = __HAL_TIM_GET_AUTORELOAD(&htim2);
 SetDutyCycle = AutoReload * Duty / 100.0;
 __HAL_TIM_SET_COMPARE(&htim2, Channel, SetDutyCycle);
}
```

- Inside the main program, the three channels are started and their duty cycles are set.

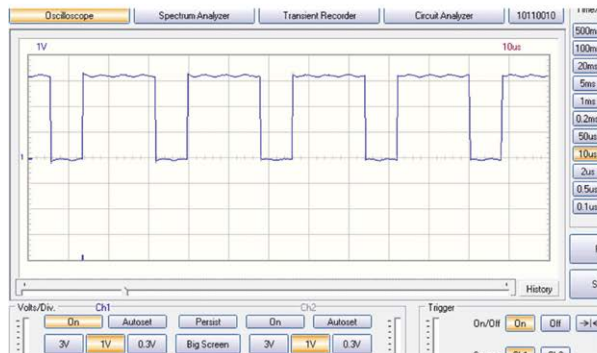
Figure 10.14 shows the generated waveforms for each channel. Here, the vertical axis was 1 V/division and the horizontal axis was 10  $\mu$ s/division.



**Pin: PA15, Duty Cycle = 25%**



**Pin: PB3, Duty Cycle = 50%**



**Pin: PB10, Duty Cycle = 75%**

*Figure 10.14: Generated waveforms.*

The complete program listing is shown in Figure 10.15.

```
/* USER CODE BEGIN Header */
/**

 * @file : main.c
 * @brief : Main program body

 * @attention
 *
 * <h2><center>© Copyright (c) 2020 STMicroelectronics.
 * All rights reserved.</center></h2>
 *
 * This software component is licensed by ST under BSD 3-Clause license,
 * the «License»; You may not use this file except in compliance with the
 * License. You may obtain a copy of the License at:
 *
 * opensource.org/licenses/BSD-3-Clause
 *

 */
#include «main.h»

TIM_HandleTypeDef htim2;

void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_TIM2_Init(void);

//
// This function sets the Duty Cycle (in %) for a given Channel of TIM2
//
void DutyCycle(float Duty, int Channel)
{
 uint16_t AutoReload, SetDutyCycle;
 AutoReload = __HAL_TIM_GET_AUTORELOAD(&htim2);
 SetDutyCycle = AutoReload * Duty / 100.0;
 __HAL_TIM_SET_COMPARE(&htim2, Channel, SetDutyCycle);
}

int main(void)
{
 HAL_Init();

 SystemClock_Config();

 MX_GPIO_Init();
```

```

MX_TIM2_Init();

HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_1); // Start Channel 1
HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_2); // Start Channel 2
HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_3); // Start Channel 3

DutyCycle(25, TIM_CHANNEL_1); // Duty cycle=25%
DutyCycle(50, TIM_CHANNEL_2); // Duty cycle=50%
DutyCycle(75, TIM_CHANNEL_3); // Duty cycle=75%

while (1)
{

}

}

void SystemClock_Config(void)
{
 RCC_OscInitTypeDef RCC_OscInitStruct = {0};
 RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

 RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
 RCC_OscInitStruct.HSISState = RCC_HSI_ON;
 RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
 RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
 RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
 RCC_OscInitStruct.PLL.PLLM = 2;
 RCC_OscInitStruct.PLL.PLLN = 20;
 RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV7;
 RCC_OscInitStruct.PLL.PLLQ = RCC_PLLQ_DIV2;
 RCC_OscInitStruct.PLL.PLLR = RCC_PLLR_DIV2;
 if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
 {
 Error_Handler();
 }

 RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
 |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
 RCC_ClkInitStruct.SYSClkSource = RCC_SYSCCLKSOURCE_PLLCLK;
 RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCCLK_DIV1;
 RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
 RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

 if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_4) != HAL_OK)
 {
 Error_Handler();
 }
}

```

```
}

if (HAL_PWREx_ControlVoltageScaling(PWR_REGULATOR_VOLTAGE_SCALE1) != HAL_OK)
{
 Error_Handler();
}
}

static void MX_TIM2_Init(void)
{
 TIM_ClockConfigTypeDef sClockSourceConfig = {0};
 TIM_MasterConfigTypeDef sMasterConfig = {0};
 TIM_OC_InitTypeDef sConfigOC = {0};

 htim2.Instance = TIM2;
 htim2.Init.Prescaler = 80;
 htim2.Init.CounterMode = TIM_COUNTERMODE_UP;
 htim2.Init.Period = 25;
 htim2.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
 htim2.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_ENABLE;
 if (HAL_TIM_Base_Init(&htim2) != HAL_OK)
 {
 Error_Handler();
 }
 sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
 if (HAL_TIM_ConfigClockSource(&htim2, &sClockSourceConfig) != HAL_OK)
 {
 Error_Handler();
 }
 if (HAL_TIM_PWM_Init(&htim2) != HAL_OK)
 {
 Error_Handler();
 }
 sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
 sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
 if (HAL_TIMEx_MasterConfigSynchronization(&htim2, &sMasterConfig) != HAL_OK)
 {
 Error_Handler();
 }
 sConfigOC.OCMode = TIM_OCMODE_PWM1;
 sConfigOC.Pulse = 0;
 sConfigOC.OCpolarity = TIM_OCPOLARITY_HIGH;
 sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
 if (HAL_TIM_PWM_ConfigChannel(&htim2, &sConfigOC, TIM_CHANNEL_1) != HAL_OK)
 {
 Error_Handler();
 }
}
```

```

 }
 if (HAL_TIM_PWM_ConfigChannel(&htim2, &sConfigOC, TIM_CHANNEL_2) != HAL_OK)
 {
 Error_Handler();
 }
 if (HAL_TIM_PWM_ConfigChannel(&htim2, &sConfigOC, TIM_CHANNEL_3) != HAL_OK)
 {
 Error_Handler();
 }

 HAL_TIM_MspPostInit(&htim2);
}

static void MX_GPIO_Init(void)
{
 /* GPIO Ports Clock Enable */
 __HAL_RCC_GPIOB_CLK_ENABLE();
 __HAL_RCC_GPIOA_CLK_ENABLE();
}

void Error_Handler(void)
{
}

#ifdef USE_FULL_ASSERT

void assert_failed(uint8_t *file, uint32_t line)
{
}

#endif

/***** (C) COPYRIGHT STMicroelectronics *****/

```

*Figure 10.15: The PWM3 program.*

### 10.7 Project 4: Potentiometer-controlled Duty Cycle Control of PWM Waveform. Description

In this project a PWM waveform with the frequency of 1 kHz is generated at port pin PA15, where an LED is connected to this port pin. A potentiometer is connected to analogue input PC2 (analogue input IN3) of the development board. By moving the potentiometer arm, the duty cycle of the PWM waveform is changed from 0% to 100%. As a result, the brightness of the LED changes from OFF to full brightness.



## Block diagram

Figure 10.16 shows the block diagram of the project.

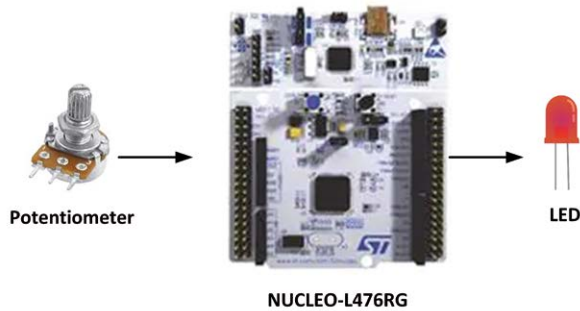


Figure 10.16: Block diagram of the project.

## Circuit diagram

The circuit diagram of the project is shown in Figure 10.17. The LED is connected to PA15 through a 470 Ohm current limiting resistor. The arm of a 10-kohm potentiometer is connected to pin PC2, while the other two terminals are connected to +3.3 V and ground.

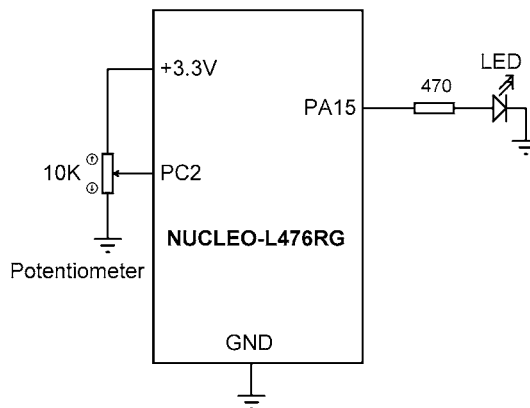


Figure 10.17: Circuit diagram of the project.

## Program listing

In this project, Channel 2 of timer TIM2 is used to generate the PWM waveform. The steps are as follows.

- Start the STM32CubeIDE program.
- Select the Nucleo-L476RG board as before.
- Name the project as **PWMLED**.
- Click on pin **PA15** and enable **TIM2\_CH1**.
- Click **TIM2** under **Timers** and select **Internal Clock** as the **Clock Source**, enable **PWM Generation CH1**.
- In **Parameter Settings**, set the **Counter Mode** to **Up**.

- We now have to set the timer parameters for the required PWM specifications. The required PWM frequency is 1 kHz, i.e. period is 1.0 ms (1000  $\mu$ s) and the required PWM pulsewidth is 0.5 ms (500  $\mu$ s, i.e. 50% duty cycle). The timer clock frequency is set to 80 MHz (period = 0.0125  $\mu$ s). If we set the prescaler to 80, then  $80 \times 0.0125 = 1 \mu$ s. Therefore, for a period of 1000  $\mu$ s, the counter should be set to 1000. Enable **auto-reload preload**. Leave the other settings as they are.
- On the pin layout click **PC2** and select **ADC1\_IN3**.
- Click **Analog** and select **ADC1**.
- Click **IN3** and set to **IN3 Single-ended**.
- In **Parameters Settings**, set **Resolution** to **ADC 12-bit resolution**, enable **Continuous Conversion mode**, set **Overrun behaviour** to **Overrun data overwritten**.
- Configure for 80 MHz clock.
- Click **File**, followed by **Save**, and click **YES** to generate code.
- Click **Core**, followed by **Src**, then double click **main.c** to display the main program.

Function DutyCycle developed in the previous project is also used in this project to change the duty cycle. Start the ADC and the timer using the following functions:

```
HAL_ADC_Start(&hadc1);
HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_1);
```

As the potentiometer arm is moved from one side to the other side the analogue voltage at pin PC2 changes from 0 to +3.3 V, or from 0 to 4095 as digital value. The range of the duty cycle is from 0% to 100%. We therefore should map numbers (0 to 4095) to (0 to 100), so that as we move the potentiometer wiper (arm), the duty cycle varies from 0% to 100%. What is required is linear mapping and it can be done as follows:

Let the output\_start and output\_end numbers to be Os and Oe.

Also, let the input\_start and input\_end numbers to be Is and Ie.

If I and O are the input and output numbers respectively, the required output number is given by:

$$O = O_s + ((O_e - O_s) / (I_e - I_s)) \times (I - I_s)$$

Where,

$$O_s = 0$$

$$O_e = 100$$

$$I_s = 0$$

$$I_e = 4095$$

or,

$$O = 100 \times I / 4095$$

Where, O is the duty cycle, and I is the analogue value read from the ADC

Some example values are given below:

| <b>Analog value</b> | <b>Duty cycle</b> |
|---------------------|-------------------|
| 100                 | 2.44              |
| 200                 | 4.88              |
| 500                 | 12.2              |
| 1000                | 24.4              |
| 2000                | 48.8              |
| 3000                | 73.2              |
| 4000                | 97.6              |
| 4095                | 100               |

The mapping is implemented inside the program loop which has the following statements:

```
while (1)
{
 HAL_ADC_PollForConversion(&hadc1, 100);
 adcResult = HAL_ADC_GetValue(&hadc1);
 duty = 100.0 * adcResult / 4095.0;
 DutyCycle(duty, TIM_CHANNEL_1);
}
```

You will notice that as the potentiometer arm is moved the brightness of the LED changes. Figure 10.18 shows the program listing.

```
/* USER CODE BEGIN Header */
/**
 * *****
 * @file : main.c
 * @brief : Main program body
 * *****
 * @attention
 *
 * <h2><center>© Copyright (c) 2020 STMicroelectronics.
 * All rights reserved.</center></h2>
 *
 * This software component is licensed by ST under BSD 3-Clause license,
 * the «License»; You may not use this file except in compliance with the
 * License. You may obtain a copy of the License at:
 *
 * opensource.org/licenses/BSD-3-Clause
 *
 * *****
 */

#include «main.h»
#define LED GPIO_PIN_15
```

```

ADC_HandleTypeDef hadc1;

TIM_HandleTypeDef htim2;

void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_ADC1_Init(void);
static void MX_TIM2_Init(void);

//
// This function sets the Duty Cycle (in %) for a given Channel of TIM2
//
void DutyCycle(float Duty, int Channel)
{
 uint16_t AutoReload, SetDutyCycle;
 AutoReload = __HAL_TIM_GET_AUTORELOAD(&htim2);
 SetDutyCycle = AutoReload * Duty / 100.0;
 __HAL_TIM_SET_COMPARE(&htim2, Channel, SetDutyCycle);
}

int main(void)
{
 uint32_t adcResult;
 float duty;

 HAL_Init();

 SystemClock_Config();

 MX_GPIO_Init();
 MX_ADC1_Init();
 MX_TIM2_Init();

 HAL_ADC_Start(&hadc1); // Start ADC
 HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_1); // Start Channel 1

 while (1)
 {
 HAL_ADC_PollForConversion(&hadc1, 100); // Wait for conversion
 adcResult = HAL_ADC_GetValue(&hadc1); // Get analog data
 duty = 100.0 * adcResult / 4095.0; // Calculate duty cycle
 DutyCycle(duty, TIM_CHANNEL_1); // Set duty cycle
 }
}

```

```
void SystemClock_Config(void)
{
 RCC_OscInitTypeDef RCC_OscInitStruct = {0};
 RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};
 RCC_PeriphCLKInitTypeDef PeriphClkInit = {0};

 RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
 RCC_OscInitStruct.HSISState = RCC_HSI_ON;
 RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
 RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
 RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
 RCC_OscInitStruct.PLL.PLLM = 2;
 RCC_OscInitStruct.PLL.PLLN = 20;
 RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV7;
 RCC_OscInitStruct.PLL.PLLQ = RCC_PLLQ_DIV2;
 RCC_OscInitStruct.PLL.PLLR = RCC_PLLR_DIV2;
 if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
 {
 Error_Handler();
 }

 RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
 |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
 RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
 RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
 RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
 RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

 if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_4) != HAL_OK)
 {
 Error_Handler();
 }
 PeriphClkInit.PeriphClockSelection = RCC_PERIPHCLK_ADC;
 PeriphClkInit.AdcClockSelection = RCC_ADCCLKSOURCE_PLLSAI1;
 PeriphClkInit.PLLSAI1.PLLSAI1Source = RCC_PLLSOURCE_HSI;
 PeriphClkInit.PLLSAI1.PLLSAI1M = 2;
 PeriphClkInit.PLLSAI1.PLLSAI1N = 8;
 PeriphClkInit.PLLSAI1.PLLSAI1P = RCC_PLLP_DIV7;
 PeriphClkInit.PLLSAI1.PLLSAI1Q = RCC_PLLQ_DIV2;
 PeriphClkInit.PLLSAI1.PLLSAI1R = RCC_PLLR_DIV2;
 PeriphClkInit.PLLSAI1.PLLSAI1ClockOut = RCC_PLLSAI1_ADC1CLK;
 if (HAL_RCCEx_PeriphCLKConfig(&PeriphClkInit) != HAL_OK)
 {
 Error_Handler();
 }
}
```

```

 if (HAL_PWREx_ControlVoltageScaling(PWR_REGULATOR_VOLTAGE_SCALE1) != HAL_OK)
 {
 Error_Handler();
 }
}

static void MX_ADC1_Init(void)
{
 ADC_MultiModeTypeDef multimode = {0};
 ADC_ChannelConfTypeDef sConfig = {0};

 hadc1.Instance = ADC1;
 hadc1.Init.ClockPrescaler = ADC_CLOCK_ASYNC_DIV1;
 hadc1.Init.Resolution = ADC_RESOLUTION_12B;
 hadc1.Init.DataAlign = ADC_DATAALIGN_RIGHT;
 hadc1.Init.ScanConvMode = ADC_SCAN_DISABLE;
 hadc1.Init.EOCSelection = ADC_EOC_SINGLE_CONV;
 hadc1.Init.LowPowerAutoWait = DISABLE;
 hadc1.Init.ContinuousConvMode = ENABLE;
 hadc1.Init.NbrOfConversion = 1;
 hadc1.Init.DiscontinuousConvMode = DISABLE;
 hadc1.Init.NbrOfDiscConversion = 1;
 hadc1.Init.ExternalTrigConv = ADC_SOFTWARE_START;
 hadc1.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_NONE;
 hadc1.Init.DMAContinuousRequests = DISABLE;
 hadc1.Init.Overrun = ADC_OVR_DATA_OVERWRITTEN;
 hadc1.Init.OversamplingMode = DISABLE;
 if (HAL_ADC_Init(&hadc1) != HAL_OK)
 {
 Error_Handler();
 }

 multimode.Mode = ADC_MODE_INDEPENDENT;
 if (HAL_ADCEx_MultiModeConfigChannel(&hadc1, &multimode) != HAL_OK)
 {
 Error_Handler();
 }

 sConfig.Channel = ADC_CHANNEL_3;
 sConfig.Rank = ADC_REGULAR_RANK_1;
 sConfig.SamplingTime = ADC_SAMPLETIME_2CYCLES_5;
 sConfig.SingleDiff = ADC_SINGLE_ENDED;
 sConfig.OffsetNumber = ADC_OFFSET_NONE;
 sConfig.Offset = 0;
 if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)

```

```
{
 Error_Handler();
}
}

static void MX_TIM2_Init(void)
{
 TIM_ClockConfigTypeDef sClockSourceConfig = {0};
 TIM_MasterConfigTypeDef sMasterConfig = {0};
 TIM_OC_InitTypeDef sConfigOC = {0};

 htim2.Instance = TIM2;
 htim2.Init.Prescaler = 80;
 htim2.Init.CounterMode = TIM_COUNTERMODE_UP;
 htim2.Init.Period = 1000;
 htim2.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
 htim2.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_ENABLE;
 if (HAL_TIM_Base_Init(&htim2) != HAL_OK)
 {
 Error_Handler();
 }
 sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
 if (HAL_TIM_ConfigClockSource(&htim2, &sClockSourceConfig) != HAL_OK)
 {
 Error_Handler();
 }
 if (HAL_TIM_PWM_Init(&htim2) != HAL_OK)
 {
 Error_Handler();
 }
 sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
 sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
 if (HAL_TIMEx_MasterConfigSynchronization(&htim2, &sMasterConfig) != HAL_OK)
 {
 Error_Handler();
 }
 sConfigOC.OCMode = TIM_OCMODE_PWM1;
 sConfigOC.Pulse = 0;
 sConfigOC.OCpolarity = TIM_OCPOLARITY_HIGH;
 sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
 if (HAL_TIM_PWM_ConfigChannel(&htim2, &sConfigOC, TIM_CHANNEL_1) != HAL_OK)
 {
 Error_Handler();
 }
}
```

```

 HAL_TIM_MspPostInit(&htim2);
}

static void MX_GPIO_Init(void)
{
 /* GPIO Ports Clock Enable */
 __HAL_RCC_GPIOC_CLK_ENABLE();
 __HAL_RCC_GPIOA_CLK_ENABLE();
}

void Error_Handler(void)
{
}

#ifdef USE_FULL_ASSERT

void assert_failed(uint8_t *file, uint32_t line)
{
}

#endif

/***** (C) COPYRIGHT STMicroelectronics *****/

```

*Figure 10.18: The PWMLED program.*

### 10.8 Summary

In this Chapter we have learned how to generate PWM waveforms. Several projects were given that effect. In the next Chapter we will be focusing on the serial communication where we will be using the UART ports to send and receive serial data using our Nucleo development board with the STM32CubeIDE software.



## CHAPTER 11 • Serial Communication

### 11.1 Overview

Serial communication is a simple means of sending data across long distances quickly and reliably. The most commonly used serial communication method is based on the historic RS232 standard. In this standard, data is sent over a single line from a transmitting device to a receiving device in bit serial format at a prespecified speed, also known as the Baud rate, or the number of bits sent each second. Typical Baud rates expressed in bits per second are 4800, 9600, 19200, 38400, etc.

RS232 serial communication is a form of asynchronous data transmission where data is sent character by character. Each character is preceded with a Start bit, seven or eight data bits, an optional parity bit, and one or more stop bits. The most commonly used format is eight data bits, no parity bit and one stop bit. The least significant data bit is transmitted first, and the most significant bit is transmitted last.

In RS232 communication, a logic high is defined to be at  $-12\text{ V}$ , and a logic 0, at  $+12\text{ V}$ . Nowadays, serial communication is used with TTL level signals where a logic high is defined as  $+5\text{ V}$ , and logic 0 is defined as  $0\text{ V}$ . Figure 11.1 shows how character 'A' (ASCII binary pattern 0010 0001) is transmitted over a serial line. The line is normally idle at logic 0. The start bit is first sent by the line going from high to low. Then eight data bits are sent starting from the least significant bit. Finally, the stop bit is sent by raising the line from low to high.

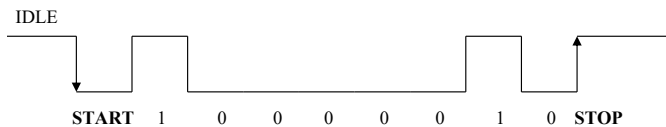


Figure 11.1: Sending character 'A' in serial format.

In serial connection, a minimum of three lines are used for communication: transmit (TX), receive (RX), and ground (GND). Some high-speed serial communication systems use additional control signals for synchronization, such as CTS, DTR, and so on. Some systems use software synchronization techniques where a special character (XOFF) is used to tell the sender to stop sending, and another character (XON) is used to tell the sender to re-start transmission.

RS232 based serial devices are connected to each other using two types of connectors: 9-way connector, and 25-way connector. Table 11.1 shows the TX, RX, and GND pins of each types of connectors. The connectors used in RS232 serial communication are shown in Figure 11.2.

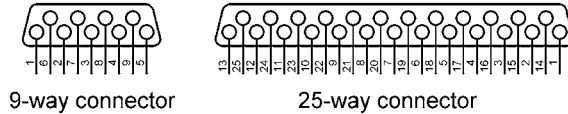
**9-pin D-sub connector**

| Pin | Function      |
|-----|---------------|
| 2   | Transmit (TX) |
| 3   | Receive (RX)  |
| 5   | Ground (GND)  |

**25-pin D-sub connector**

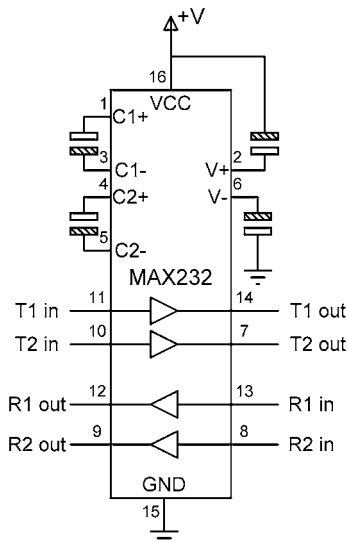
| Pin | Function      |
|-----|---------------|
| 2   | Transmit (TX) |
| 3   | Receive (RX)  |
| 7   | Ground (GND)  |

*Table 11.1: Minimum required pins for RS232 serial communication.*



*Figure 11.2: RS232 connectors.*

As described above, the RS232 voltage levels are normally at  $\pm 12$  V. However, microcontroller input-output ports operate at 0 to +5 V voltage levels. It is therefore necessary to translate the voltage levels before a microcontroller can be connected to a RS232 compatible device. Thus, the output signal from the microcontroller has to be converted into  $\pm 12$  V, and the input from an RS232 device must be converted into 0 to +5 V before it can be connected to a microcontroller. This voltage translation is normally done using special RS232 voltage converter chips. One such popular chip is the MAX232. This is a dual converter chip having the pin configuration as shown in Figure 11.3. This particular device requires four external 1- $\mu$ F capacitors for its operation.



*Figure 11.3: MAX232 pin configuration.*

Nowadays, serial communication is done using TTL logic levels and there is no need to use  $\pm 12$  V voltage levels unless it is required to use the RS232 interface.

Serial communication can be done in either hardware or software. With the software option, all the serial bit timing is handled in software and any input-output pin can be programmed and used for serial communication. Hardware based communication uses UART (Universal Asynchronous Receiver Transmitter) or USART (USART is similar to UART but it is clock based) devices where all the data transmission is handled by the hardware. Hardware based serial communication has the advantages that it is faster, more accurate, and requires less intervention from the CPU.

Before developing UART based projects it is worthwhile to look at the various parameters used while configuring a serial port:

**Word length:** this is the number of data bits transmitted or received in a frame. Valid values are 7 and 8. In most serial communication applications 8 data bits are used.

**Stop bits:** this field specifies the number of stop bits used. Valid values are 1 and 2. In most serial communication applications 1 stop bit is used.

**Parity:** this bit is used for error checking when a byte is received or transmitted. Valid values are: EVEN, ODD, and NONE. In most serial communication applications NONE is used.

**Baud rate:** this is the speed of data transmission in bits per second. In most applications a frame consists of a start bit, 8 data bits, and a stop bit. i.e. 10 bits are used to transmit a byte. Typical baud rates are 9600, 19200, 38400, 76800 and so on. For example, at 9600 baud rate, 960 bytes can be transmitted or received every second.

**Flow control:** in some high speed applications it may be required to use flow control bits to synchronize the sending and receiving ends. The flow control can be implemented in software (e.g. handshaking) or in hardware (e.g. using CTS and RTS lines). In the projects featured in this book we will not be using flow control.

In this Chapter we will be developing projects using the UARTs of the Nucleo-L476RG development board.

## 11.2 UART ports of the Nucleo-L476RG development board

The Nucleo-L476RG development board provides several UART ports. UART port 2 (Serial2 TX, Serial2 RX) is linked to the STLINK interface. When ST-LINK drivers is installed, a Virtual COM port is also installed that allows us to access the MCU UART using the USB interface. This port is used to communicate with the PC to upload our binary program to the MCU. STM32L476RG MCU provides three USARTs and two UARTs. All of these UARTs/USARTs support hardware flow control:

| UART/USART | GPIO pin |                              |
|------------|----------|------------------------------|
| USART 1 RX | PA10     |                              |
| USART 1 TX | PA9      |                              |
| USART 2 RX | PA3      | (Reserved for USB interface) |
| USART 2 TX | PA2      | (Reserved for USB interface) |
| USART 3 RX | PC5      |                              |

|            |      |
|------------|------|
| USART 3 TX | PC4  |
| UART 4 RX  | PA1  |
| UART 4 TX  | PA0  |
| UART 5 RX  | PD2  |
| UART 5 TX  | PC12 |

### 11.3 Serial communication program on a PC

In order to communicate with a PC over a serial line we need to install a terminal emulator program on our PC. There are several freely available terminal emulation programs available on the Internet, such as HyperTrm, Putty, TeraTerm and so on.

Before we use a terminal emulation program to communicate over a serial port we need to know the following information about the device that we wish to communicate with:

- assigned serial port number;
- speed of communication (i.e. Baud rate);
- databits (7 or 8);
- parity (even, odd, or none);
- stop bits (1 or 2);
- handshaking.

The PC uses the USB port as a serial port. The serial port number can be obtained as follows (Windows 10 system).

- Plug-in your Nucleo-L476RG board to the PC using the USB cable.
- Click **Start** and then click on the **Control Panel**.
- Click on **System** and then click on **Device Manager**.
- Click to expand **Ports (COM & LPT)** and check the **STMicroelectronics STLink Virtual COM Port** number. As an example, in Figure 11.4 the port number was **COM6**.

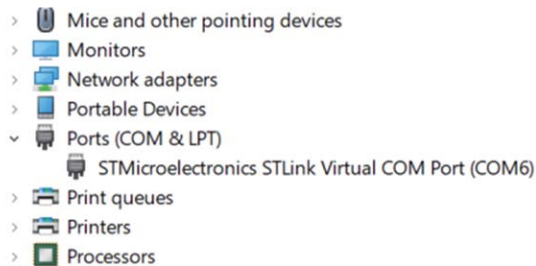


Figure 11.4: Assigned serial port number.

#### Configuring HyperTrm

The steps are as follows.

- Start HyperTrm.
- Enter a name for the connection, e.g. Nucleo and click **OK**.
- Select the serial port, e.g. COM6 (Figure 11.5) and click **OK**.



Figure 11.5: Selecting the serial port.

- Set the communication parameters, e.g.: baud rate to 9600, 8 data bits, parity None, 1 stop bit, None flow control (Figure 11.6) and click **OK**.

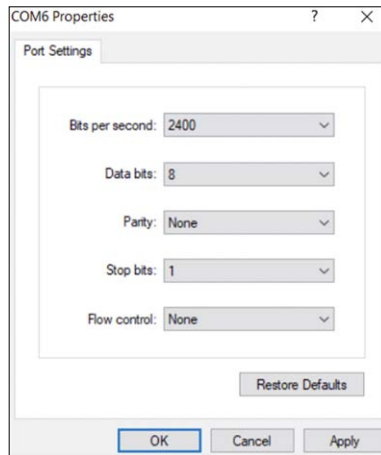


Figure 11.6: Setting the communication parameters.

- You should see the terminal emulation screen.

## Configuring Putty

The steps are as follows.

- Start Putty.
- Click **Serial** and set port number, e.g. COM6, set the baud rate to 9600 (Figure 11.7).

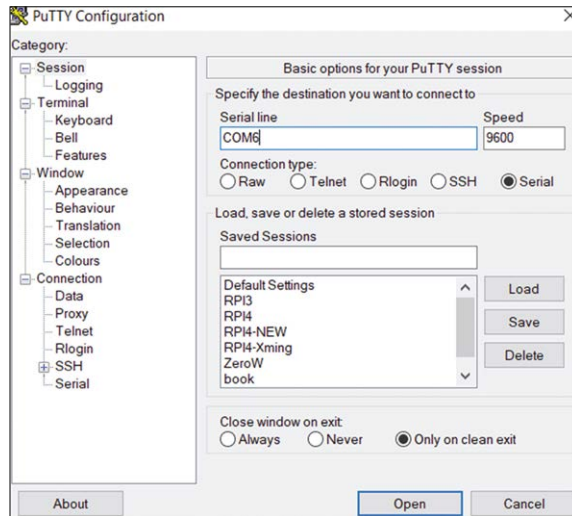


Figure 11.7: Configuring Putty.

- Click **Open** and you should see the terminal emulation screen.

A nice feature of Putty is that the communication parameters and screen settings (e.g. text size, text and background colour, bold etc) can be saved and then loaded after starting Putty.

Suppose that we want to configure Putty for the following settings and save these settings in a file called Nucleo.

|                    |                  |
|--------------------|------------------|
| Mode:              | serial           |
| Port:              | COM6             |
| Baud rate:         | 9600             |
| Text colour:       | black            |
| Background colour: | white            |
| Text font:         | Courier New      |
| Text size:         | 12 point regular |
| Cursor colour:     | black            |
| Session name:      | Nucleo           |

The steps are as follows.

- Start Putty.
- Click **Serial** and set port number to COM6, set the baud rate to 9600.
- Click **Colours** under **Window**.
- Set **Default Foreground** to black, i.e. Red=0, Green=0, Blue=0.
- Set **Default Bold Foreground** to black, i.e. Red=0, Green=0, Blue=0.
- Set **Default Background** to white, i.e. Red=255, Green=255, Blue=255.
- Set **Default Bold Background** to white, i.e. Red=255, Green=255, Blue=255.
- Set **Cursor Text** to black, i.e. Red=0, Green=0, Blue=0.
- Click **Appearance** under **Window**.

- Click **Change** and select **Courier New Regular Size 12**.
- Click **Session**.
- Enter **Nucleo** to **Saved Sessions** and click **Save**.
- Click **Open** to open the terminal emulator.

Next time you start Putty, select **Nucleo**, followed by **Load**, and click **Open** to start the session.

## 11.4 Project 1: Displaying Text on the PC

### Description

In this simple project we will send the text 'Hello from Nucleo-L476RG' to your PC.

### The aim

The aim of this project is to show how the USART can be used to send text message to the PC.

### Program listing

In this program we will be using USART 2 (at pins PA2 and PA3) which is the USB serial port connected to the PC.

The steps are as follows.

- Start STM32CubeIDE.
- Create a new working directory.
- Create a new STM32 project.
- Select STM32L476RG as the MCU.
- Give the name **PCTEXT** to the program and click **YES**.
- Select **USART2** under tab **Connectivity** and set its **Mode** to **Asynchronous** (Figure 11.8).

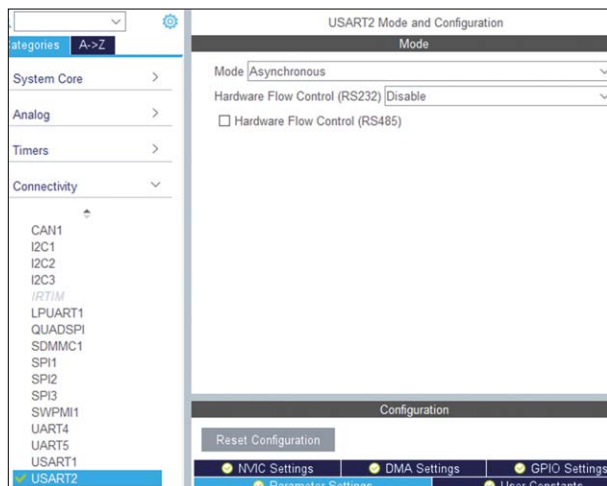


Figure 11.8: Select USART2.

- In **Parameter Settings**, set the **Baud Rate** to 9600, **Word Length** to 8, **Parity** to None, and **Stop Bits** to 1.
- Configure the MCO for 80 MHz clock.
- Click **File**, followed by **Save**, and click **YES** to generate code.
- Click **Core**, followed by **Src**, and double click **main.h** to open the main program.

You will notice that the following new lines are in the program related to USART:

```
UART_HandleTypeDef huart2;
static void MX_USART2_UART_Init(void);
MX_USART2_UART_Init(void)
```

The following function is added to configure the UART parameters:

```
static void MX_USART2_UART_Init(void)
{
 huart2.Instance = USART2;
 huart2.Init.BaudRate = 9600;
 huart2.Init.WordLength = UART_WORDLENGTH_8B;
 huart2.Init.StopBits = UART_STOPBITS_1;
 huart2.Init.Parity = UART_PARITY_NONE;
 huart2.Init.Mode = UART_MODE_TX_RX;
 huart2.Init.HwFlowCtl = UART_HWCONTROL_NONE;
 huart2.Init.OverSampling = UART_OVERSAMPLING_16;
 huart2.Init.OneBitSampling = UART_ONE_BIT_SAMPLE_DISABLE;
 huart2.AdvancedInit.AdvFeatureInit = UART_ADVFEATURE_NO_INIT;
 if (HAL_UART_Init(&huart2) != HAL_OK)
 {
 Error_Handler();
 }
}
```

We have to include the header file **string.h** at the beginning of the program. Also, the following function should be added to the program to send text message to the PC:

```
void UART_SEND(UART_HandleTypeDef *huart, char buffer[])
{
 HAL_UART_Transmit(huart, (uint8_t*) buffer, strlen(buffer), HAL_MAX_DELAY);
}
```

Enter the following statement before the program loop to display the required text on the PC:

```
UART_SEND(&huart2, "Hello from Nucleo-L476RG");
```



Open a terminal emulation session. Compile the program in **Release** mode, making sure there are no errors. Drag and drop the binary file **PCTEXT.bin** to device NUCLEO\_L476RG. You should see the text displayed on the PC as shown in Figure 11.9 (in this project, HyperTrm is used).

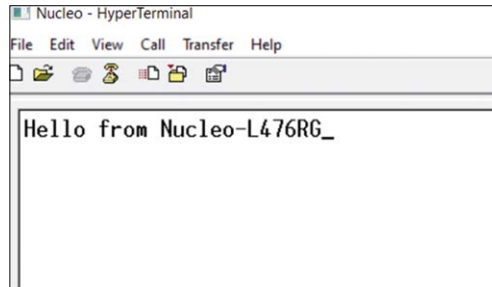


Figure 11.9: Displaying the text on terminal emulator screen.

Figure 11.10 shows the program listing.

```
/* USER CODE BEGIN Header */
/**
 * *****
 * @file : main.c
 * @brief : Main program body
 * *****
 * @attention
 *
 * <h2><center>© Copyright (c) 2020 STMicroelectronics.
 * All rights reserved.</center></h2>
 *
 * This software component is licensed by ST under BSD 3-Clause license,
 * the «License»; You may not use this file except in compliance with the
 * License. You may obtain a copy of the License at:
 *
 * opensource.org/licenses/BSD-3-Clause
 *
 * *****
 */
#include «main.h»
#include «string.h»

UART_HandleTypeDef huart2;

void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_USART2_UART_Init(void);

//
```

```

// Send data to UART
//
void UART_SEND(UART_HandleTypeDef *huart, char buffer[])
{
 HAL_UART_Transmit(huart, (uint8_t*) buffer, strlen(buffer), HAL_MAX_DELAY);
}

//
// Start of main program
//
int main(void)
{
 HAL_Init();

 SystemClock_Config();

 MX_GPIO_Init();
 MX_USART2_UART_Init();

 UART_SEND(&huart2, «Hello from Nucleo-L476RG»);

 while (1)
 {

 }
}

void SystemClock_Config(void)
{
 RCC_OscInitTypeDef RCC_OscInitStruct = {0};
 RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};
 RCC_PeriphCLKInitTypeDef PeriphClkInit = {0};

 RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
 RCC_OscInitStruct.HSISState = RCC_HSI_ON;
 RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
 RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
 RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
 RCC_OscInitStruct.PLL.PLLM = 2;
 RCC_OscInitStruct.PLL.PLLN = 20;
 RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV7;
 RCC_OscInitStruct.PLL.PLLQ = RCC_PLLQ_DIV2;
 RCC_OscInitStruct.PLL.PLLR = RCC_PLLR_DIV2;
 if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
 {

```

```
Error_Handler();
}

RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
 |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_4) != HAL_OK)
{
 Error_Handler();
}
PeriphClkInit.PeriphClockSelection = RCC_PERIPHCLK_USART2;
PeriphClkInit.Usart2ClockSelection = RCC_USART2CLKSOURCE_PCLK1;
if (HAL_RCCEx_PeriphCLKConfig(&PeriphClkInit) != HAL_OK)
{
 Error_Handler();
}

if (HAL_PWREx_ControlVoltageScaling(PWR_REGULATOR_VOLTAGE_SCALE1) != HAL_OK)
{
 Error_Handler();
}
}

static void MX_USART2_UART_Init(void)
{
 huart2.Instance = USART2;
 huart2.Init.BaudRate = 9600;
 huart2.Init.WordLength = UART_WORDLENGTH_8B;
 huart2.Init.StopBits = UART_STOPBITS_1;
 huart2.Init.Parity = UART_PARITY_NONE;
 huart2.Init.Mode = UART_MODE_TX_RX;
 huart2.Init.HwFlowCtl = UART_HWCONTROL_NONE;
 huart2.Init.OverSampling = UART_OVERSAMPLING_16;
 huart2.Init.OneBitSampling = UART_ONE_BIT_SAMPLE_DISABLE;
 huart2.AdvancedInit.AdvFeatureInit = UART_ADVFEATURE_NO_INIT;
 if (HAL_UART_Init(&huart2) != HAL_OK)
 {
 Error_Handler();
 }
}
```

```

static void MX_GPIO_Init(void)
{
 /* GPIO Ports Clock Enable */
 __HAL_RCC_GPIOA_CLK_ENABLE();
}

void Error_Handler(void)
{

}

#ifdef USE_FULL_ASSERT

void assert_failed(uint8_t *file, uint32_t line)
{

}

#endif

/***** (C) COPYRIGHT STMicroelectronics *****/

```

*Figure 11.10: Program: PCTEXT.*

## 11.5 Project 2: Simple Up Counter

### Description

In this project we will create an up counter and send the counts to the PC every second.

### The aim

The aim of this project is to show how numeric data with carriage-return (CR) and line-feed (LF) can be sent to a PC.

### Program listing

The program is same as the one given in Figure 11.10, except that the main program is modified and is shown in Figure 11.11. Here, variable Count increments every second. It is converted into string by calling built-in function **itoa**. Function **UART\_SEND** is then called to display the count. Notice that carriage-return and line-feed characters joined to the end of the data so that each count value is displayed on a new line.

```

#include «main.h»
#include «string.h»
#include «stdlib.h»

UART_HandleTypeDef huart2;

void SystemClock_Config(void);
static void MX_GPIO_Init(void);

```

```
static void MX_USART2_UART_Init(void);

//
// Send data to UART
//
void UART_SEND(UART_HandleTypeDef *huart, char buffer[])
{
 HAL_UART_Transmit(huart, (uint8_t*) buffer, strlen(buffer), HAL_MAX_DELAY);
}

//
// Start of main program
//
int main(void)
{
 int Count = 0;
 char buffer[10];
 char CRLF[3] = {0x0A, 0x0D, 0x00};

 HAL_Init();

 SystemClock_Config();

 MX_GPIO_Init();
 MX_USART2_UART_Init();

 while (1)
 {
 itoa(Count, buffer, 10);
 strcat(buffer, CRLF);
 UART_SEND(&huart2, buffer);
 Count++;
 HAL_Delay(1000);
 }
}
```

*Figure 11.11: Program listing.*

Figure 11.12 shows part of the count on the terminal emulator window.

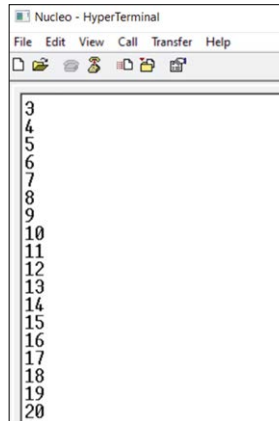


Figure 11.12: Part of the display.

We can create the following functions to make it easier to send text, numbers, and new-line to the UART. Notice that the last parameter is for new-line (CR/LF) control. Setting to '1' displays a new line.

1. To send new-line (CR/LF; carriage-return + line-feed): — Call as `UART_SEND_NL(&huart2);`

```
void UART_SEND_NL(UART_HandleTypeDef *huart)
{
 HAL_UART_Transmit(huart, (uint8_t*)"\\n\\r", 2, HAL_MAX_DELAY);
}
```

2. To send a character: — Call as `UART_SEND_CHR(&huart2, "x", 0);`

```
void UART_SEND_CHR(UART_HandleTypeDef *huart, char c, int m)
{
 HAL_UART_Transmit(huart, (uint8_t*)&c, 1, HAL_MAX_DELAY);
 if(m == 1) HAL_UART_Transmit(huart, (uint8_t*)"\\n\\r", 2, HAL_MAX_DELAY);
}
```

3. To send an integer number: — Call as `UART_SEND_INT(&huart2, Count, 0);`

```
void UART_SEND_INT(UART_HandleTypeDef *huart, int i, int m)
{
 char buffer[10];
 itoa(i, buffer, 10);
 HAL_UART_Transmit(huart, (uint8_t*) buffer, strlen(buffer), HAL_MAX_DELAY);
 if(m == 1) HAL_UART_Transmit(huart, (uint8_t*)"\\n\\r", 2, HAL_MAX_DELAY);
}
```

4. To send text: - Call as `UART_SEND_TXT(&huart2, "Test", 0);`

```
void UART_SEND_TXT(UART_HandleTypeDef *huart, char buffer[], int m)
{
 HAL_UART_Transmit(huart, (uint8_t*) buffer, strlen(buffer), HAL_MAX_DELAY);
 if(m == 1) HAL_UART_Transmit(huart, (uint8_t*)"\\n\\r", 2, HAL_MAX_DELAY);
}
```

5. To send floating point number: — Call as `UART_SEND_FLT(&huart2, q, 0);`

```
void UART_SEND_FLT(UART_HandleTypeDef *huart, float p, int m)
{
 char buffer[10];
 sprintf(buffer, "%5.2f", p);
 HAL_UART_Transmit(huart, (uint8_t*) buffer, strlen(buffer), HAL_MAX_DELAY);
 if(m == 1) HAL_UART_Transmit(huart, (uint8_t*)"\\n\\r", 2, HAL_MAX_DELAY);
}
```

Remember to include **stdio.h** at the beginning of the program, and also set the linker option to enable function sprint.

6. To send any type of number: - Call as `UART_SEND_ANY(&huart2, p, "%5.2f", 0);`

```
void UART_SEND_ANY(UART_HandleTypeDef *huart, float p, char *mode, int m)
{
 char buffer[10];
 sprintf(buffer, mode, p);
 HAL_UART_Transmit(huart, (uint8_t*) buffer, strlen(buffer), HAL_MAX_DELAY);
 if(m == 1) HAL_UART_Transmit(huart, (uint8_t*)"\\n\\r", 2, HAL_MAX_DELAY);
}
```

As an example, the program given in Figure 11.11 can be modified to use the integer function as shown in Figure 11.13.

```
#include «main.h»
#include «string.h»
#include «stdlib.h»

UART_HandleTypeDef huart2;

void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_USART2_UART_Init(void);

void UART_SEND_INT(UART_HandleTypeDef *huart, int i, int m)
```

```

{
 char buffer[10];
 itoa(i, buffer, 10);
 HAL_UART_Transmit(huart, (uint8_t*) buffer, strlen(buffer), HAL_MAX_DELAY);
 if(m == 1) HAL_UART_Transmit(huart, (uint8_t*)»\n\r», 2, HAL_MAX_DELAY);
}

//
// Start of main program
//
int main(void)
{
 int Count = 0;
 HAL_Init();

 SystemClock_Config();

 MX_GPIO_Init();
 MX_USART2_UART_Init();

 while (1)
 {
 UART_SEND_INT(&huart2, Count, 1);
 Count++;
 HAL_Delay(1000);
 }
}

```

*Figure 11.13: Modified program.***Cursor control**

In addition, we can use the following cursor control characters (these are known as the ‘VT100 cursor control codes’) to control the position of the cursor on the HyperTrm terminal emulator screen:

|          |                                     |
|----------|-------------------------------------|
| esc[H    | home the cursor (top left position) |
| esc[2J   | clear the screen                    |
| esc[r;cH | move cursor to row r, column c      |

Notice that **esc** is the ASCII escape character, having hexadecimal value 0x1B. We can create the following arrays for the cursor control:

```

char clrscr[] = {0x1B, '[', '2', 'J', 0}; // Clear the screen
char homescr[] = {0x1B, '[', 'H', 0}; // Home the cursor
char gotoscr[] = {0x1B, '[', 0x00, ';', 0x00, 'H', 0}; // Move cursor to the speci-
 fied position

```



The **gotoscr** command moves the cursor to the specified row and column positions. They are shown as 0x00 here but must be filled as required before used. For example, to move the cursor to line 3, column 4, we can use the following statement:

```
gotoscr[2] = '3';
gotoscr[4] = '4';
pc.printf(gotoscr);
```

The program given in Figure 11.13 has been modified so that before entering the program loop the screen is cleared and the cursor is set to the home position. Inside the program loop, the cursor is positioned at row 3, column 0 of the screen so that the numbers are displayed at the same point of the screen (see Figure 11.14).

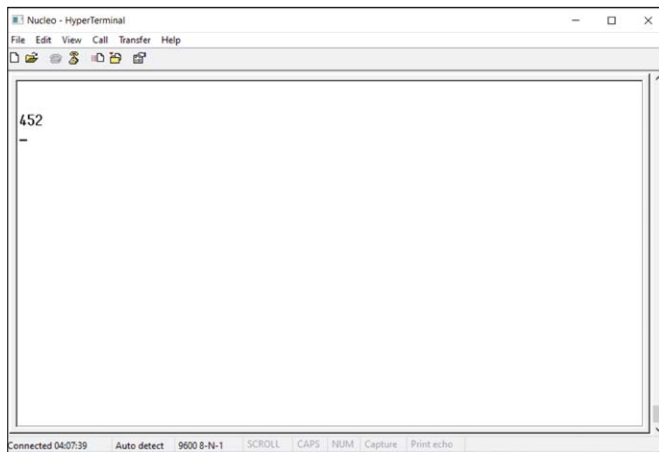


Figure 11.14: Displaying the numbers at the same point of the screen.

Figure 11.15 shows the modified program listing.

```
#include «main.h»
#include «string.h»
#include «stdlib.h»

UART_HandleTypeDef huart2;

void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_USART2_UART_Init(void);

void UART_SEND_TXT(UART_HandleTypeDef *huart, char buffer[], int m)
{
 HAL_UART_Transmit(huart, (uint8_t*) buffer, strlen(buffer), HAL_MAX_DELAY);
 if(m == 1) HAL_UART_Transmit(huart, (uint8_t*)»\n\r«, 2, HAL_MAX_DELAY);
}
```

```

void UART_SEND_INT(UART_HandleTypeDef *huart, int i, int m)
{
 char buffer[10];
 itoa(i, buffer, 10);
 HAL_UART_Transmit(huart, (uint8_t*) buffer, strlen(buffer), HAL_MAX_DELAY);
 if(m == 1) HAL_UART_Transmit(huart, (uint8_t*)»\n\r», 2, HAL_MAX_DELAY);
}

//
// Start of main program
//
int main(void)
{
 int Count = 0;

 char clrscr[] = {0x1B, '[', '2', 'J', 0};
 char homescr[] = {0x1B, '[', 'H', 0};
 char gotoscr[] = {0x1B, '[', '3', ';', '0', 'H', 0};

 HAL_Init();

 SystemClock_Config();

 MX_GPIO_Init();
 MX_USART2_UART_Init();

 UART_SEND_TXT(&huart2, clrscr, 0);
 UART_SEND_TXT(&huart2, homescr, 0);
 while (1)
 {
 UART_SEND_TXT(&huart2, gotoscr, 0);
 UART_SEND_INT(&huart2, Count, 1);
 Count++;
 HAL_Delay(1000);
 }
}

```

*Figure 11.15: Modified program listing.*

## 11.6 Project 3: Times Table

### Description

This is a times table project, where the user is prompted to enter an integer number from the keyboard. The program displays the times table (from 1 to 12) for the given number. For example, if the entered number is 5, the program will display:

```
5 × 1 = 5
5 × 2 = 10
5 × 3 = 15
5 × 4 = 20
5 × 5 = 25
5 × 6 = 30
5 × 7 = 35
5 × 8 = 40
5 × 9 = 45
5 × 10 = 50
5 × 11 = 55
5 × 12 = 60
```

### The aim

The aim of this project is to show how data can be received and sent to a PC through the UART port.

### Program listing

The clock functions, UART initialization etc are same as in the program in Figure 11.11 and therefore are not repeated here. The following user defined functions are used in the program to display text, a character, new-line, or an integer:

```
void UART_SEND_TXT(UART_HandleTypeDef *huart, char buffer[], int m)
void UART_SEND_CHR(UART_HandleTypeDef *huart, char c, int m)
void UART_SEND_NL(UART_HandleTypeDef *huart)
void UART_SEND_INT(UART_HandleTypeDef *huart, int i, int m)
```

The following function is used to receive data through the UART:

```
HAL_StatusTypeDef HAL_UART_Receive(handle, data buffer, size, Timeout);
```

The received data is stored in a buffer. It is important to notice that the function will block until all bytes specified by the size parameter are received. Set the size to 1 to receive only one byte. Timeout is the maximum time in milliseconds to wait for the receive to complete. If the transmission does not complete in the specified timeout, the function aborts and returns HAL\_TIMEOUT value, otherwise, it returns the HAL\_OK success code. We can pass HAL\_MAX\_DELAY to wait indefinitely for the receive to complete.

Figure 11.16 shows the program listing. The program displays heading **TIMES TABLE** and then prompts the user to enter a number by displaying text **Enter Number:.** The numbers entered by the user are received one at a time and stored in buffer **readBuf**, until the Enter key is detected. The received number is in character form and is converted into integer using the built-in function **atoi**. The total number is stored in integer variable **N**. Pressing the Enter key terminates the loop. The program then forms a loop and displays the times table for the required number. This process is repeated where the user is prompted to enter a number again.

The algorithm to read an integer number from the keyboard can be described by the following PDL:

```

DO FOREVER
 Read a character from the keyboard
 Echo the character on the screen
 IF Enter key is detected THEN
 Exit loop
 ENDIF
 Convert the character into a number
 Calculate the total number
ENDDO

```

The code that implements the above algorithm is shown below. At the end of this code variable **N** stores the entered integer number:

```

while(1)
{
 HAL_UART_Receive(&huart2, (uint8_t*)readBuf, 1, HAL_MAX_DELAY);
 UART_SEND_CHR(&huart2, readBuf[0], 0);
 if(readBuf[0] == '\r')break;
 n = atoi(readBuf);
 N = 10*N + n;
}

#include «main.h»
#include «string.h»
#include «stdlib.h»
#include «stdio.h»

UART_HandleTypeDef huart2;

void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_USART2_UART_Init(void);

//
// This function displays text
//
void UART_SEND_TXT(UART_HandleTypeDef *huart, char buffer[], int m)
{
 HAL_UART_Transmit(huart, (uint8_t*) buffer, strlen(buffer), HAL_MAX_DELAY);
 if(m == 1) HAL_UART_Transmit(huart, (uint8_t*)»\n\r», 2, HAL_MAX_DELAY);
}

//

```

```
// This function displays a character
//
void UART_SEND_CHR(UART_HandleTypeDef *huart, char c, int m)
{
 HAL_UART_Transmit(huart, (uint8_t*)&c, 1, HAL_MAX_DELAY);
 if(m == 1) HAL_UART_Transmit(huart, (uint8_t*)"\n\r", 2, HAL_MAX_DELAY);
}

//
// This function displays a new-line
//
void UART_SEND_NL(UART_HandleTypeDef *huart)
{
 HAL_UART_Transmit(huart, (uint8_t*)"\n\r", 2, HAL_MAX_DELAY);
}

//
// This function displays an integer number
//
void UART_SEND_INT(UART_HandleTypeDef *huart, int i, int m)
{
 char buffer[10];
 itoa(i, buffer, 10);
 HAL_UART_Transmit(huart, (uint8_t*) buffer, strlen(buffer), HAL_MAX_DELAY);
 if(m == 1) HAL_UART_Transmit(huart, (uint8_t*)"\n\r", 2, HAL_MAX_DELAY);
}

//
// Start of main program. Read an integer number from the keyboard, then
// display the times table for that number
//
int main(void)
{
 char readBuf[1];
 int N, n, p;

 HAL_Init();

 SystemClock_Config();

 MX_GPIO_Init();
 MX_USART2_UART_Init();

 while (1)
 {
 N = 0;
```

```

 UART_SEND_NL(&huart2);
 UART_SEND_TXT(&huart2, «TIMES TABLE», 1);
 UART_SEND_TXT(&huart2, «-----», 1);
 UART_SEND_NL(&huart2);
 UART_SEND_TXT(&huart2, «Enter Number: », 0);

 while(1)
 {
 HAL_UART_Receive(&huart2, (uint8_t*)readBuf, 1, HAL_MAX_DELAY);
 UART_SEND_CHR(&huart2, readBuf[0], 0);
 if(readBuf[0] == '\r')break;
 n = atoi(readBuf);
 N = 10*N + n;
 }

 UART_SEND_NL(&huart2);

 for(p = 1; p <= 12; p++)
 {
 UART_SEND_INT(&huart2, N, 0);
 UART_SEND_TXT(&huart2, « X », 0);
 UART_SEND_INT(&huart2, p, 0);
 UART_SEND_TXT(&huart2, « = », 0);
 UART_SEND_INT(&huart2, N*p, 1);
 }

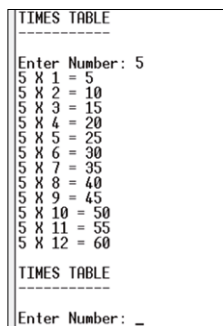
 HAL_Delay(1000);
}
}

```

Figure 11.16: Program listing.

Compile the program in **Release** mode making sure there are no errors. Drag and drop the binary file to device NUCLEO\_L476RG.

An example output from the program is shown in Figure 11.17, where the times table for 5 was requested.



```

TIMES TABLE

Enter Number: 5
5 X 1 = 5
5 X 2 = 10
5 X 3 = 15
5 X 4 = 20
5 X 5 = 25
5 X 6 = 30
5 X 7 = 35
5 X 8 = 40
5 X 9 = 45
5 X 10 = 50
5 X 11 = 55
5 X 12 = 60

TIMES TABLE

Enter Number: _

```

Figure 11.17: Times table for number 5.

## Reading an integer number

We can create a function to read and return an integer number from the keyboard until the Enter key is pressed. This function, called **ReadInt** is shown below:

```
int ReadInt(void)
{
 int n, N = 0;
 char readBuf[1];
 while(1)
 {
 HAL_UART_Receive(&huart2, (uint8_t*)readBuf, 1, HAL_MAX_DELAY);
 UART_SEND_CHR(&huart2, readBuf[0], 0);
 if(readBuf[0] == '\r')break;
 n = atoi(readBuf);
 N = 10*N + n;
 }
 return (N);
}
```

Figure 11.18 shows how this function can be used in this project.

```
#include «main.h»
#include «string.h»
#include «stdlib.h»
#include «stdio.h»

UART_HandleTypeDef huart2;

void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_USART2_UART_Init(void);

//
// This function displays text
//
void UART_SEND_TXT(UART_HandleTypeDef *huart, char buffer[], int m)
{
 HAL_UART_Transmit(huart, (uint8_t*) buffer, strlen(buffer), HAL_MAX_DELAY);
 if(m == 1) HAL_UART_Transmit(huart, (uint8_t*)»\n\r», 2, HAL_MAX_DELAY);
}

//
// This function displays a character
//
void UART_SEND_CHR(UART_HandleTypeDef *huart, char c, int m)
{

```

```

 HAL_UART_Transmit(huart, (uint8_t*)&c, 1, HAL_MAX_DELAY);
 if(m == 1) HAL_UART_Transmit(huart, (uint8_t*)»\n\r», 2, HAL_MAX_DELAY);
}

//
// This function displays a new-line
//
void UART_SEND_NL(UART_HandleTypeDef *huart)
{
 HAL_UART_Transmit(huart, (uint8_t*)»\n\r», 2, HAL_MAX_DELAY);
}

//
// This function displays an integer number
//
void UART_SEND_INT(UART_HandleTypeDef *huart, int i, int m)
{
 char buffer[10];
 itoa(i, buffer, 10);
 HAL_UART_Transmit(huart, (uint8_t*) buffer, strlen(buffer), HAL_MAX_DELAY);
 if(m == 1) HAL_UART_Transmit(huart, (uint8_t*)»\n\r», 2, HAL_MAX_DELAY);
}

//
// Read an integer number
//
int ReadInt(void)
{
 int n, N = 0;
 char readBuf[1];
 while(1)
 {
 HAL_UART_Receive(&huart2, (uint8_t*)readBuf, 1, HAL_MAX_DELAY);
 UART_SEND_CHR(&huart2, readBuf[0], 0);
 if(readBuf[0] == '\r')break;
 n = atoi(readBuf);
 N = 10*N + n;
 }
 return (N);
}

//
// Start of main program. Read an integer number from the keyboard, then
// display the times table for that number
//

```



```
int main(void)
{
 int N, p;

 HAL_Init();

 SystemClock_Config();

 MX_GPIO_Init();
 MX_USART2_UART_Init();

 while (1)
 {
 N = 0;
 UART_SEND_NL(&huart2);
 UART_SEND_TXT(&huart2, «TIMES TABLE», 1);
 UART_SEND_TXT(&huart2, «-----», 1);
 UART_SEND_NL(&huart2);
 UART_SEND_TXT(&huart2, «Enter Number: », 0);

 N = ReadInt();

 UART_SEND_NL(&huart2);

 for(p = 1; p <= 12; p++)
 {
 UART_SEND_INT(&huart2, N, 0);
 UART_SEND_TXT(&huart2, « X », 0);
 UART_SEND_INT(&huart2, p, 0);
 UART_SEND_TXT(&huart2, « = », 0);
 UART_SEND_INT(&huart2, N*p, 1);
 }
 HAL_Delay(1000);
 }
}
```

*Figure 11.18: Using the 'int' function.*

## 11.7 Project 4: Practising Elementary Multiplication

### Description

In this project the Nucleo\_L476RG processor generates two integer numbers between 1 and 100 and expects the user to calculate the multiplication of these numbers. The user enters his/her result. If the result is correct the message *Well Done!* is displayed, otherwise the message *Wrong!* ... and then the correct result is displayed.

**The aim**

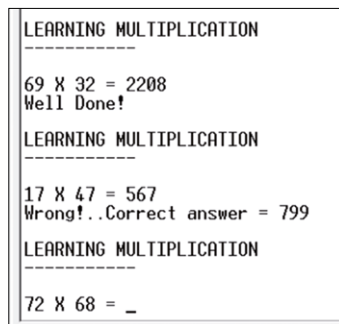
The aim of this project is to show how data can be received and sent to a PC through the UART port.

**Program listing**

The clock functions, UART initialization etc are same as in the program in Figure 11.11 and therefore are not repeated here. This program uses the random number generator to generate two integer numbers between 0 and 100.

At the beginning of the program loop the text LEARNING MULTIPLICATION is displayed. Then two random integer numbers are generated and the user is asked to calculate their product.

A typical run of the program is shown in Figure 11.19.



```

LEARNING MULTIPLICATION

69 X 32 = 2208
Well Done!

LEARNING MULTIPLICATION

17 X 47 = 567
Wrong!..Correct answer = 799

LEARNING MULTIPLICATION

72 X 68 = _

```

*Figure 11.19: Run of the program.*

Figure 11.20 shows the program listing.

```

#include «main.h»
#include «string.h»
#include «stdlib.h»
#include «stdio.h»

UART_HandleTypeDef huart2;

void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_USART2_UART_Init(void);

//
// This function displays text
//
void UART_SEND_TXT(UART_HandleTypeDef *huart, char buffer[], int m)
{
 HAL_UART_Transmit(huart, (uint8_t*) buffer, strlen(buffer), HAL_MAX_DELAY);
 if(m == 1) HAL_UART_Transmit(huart, (uint8_t*)»\n\r», 2, HAL_MAX_DELAY);
}

```

```
//
// This function displays a character
//
void UART_SEND_CHR(UART_HandleTypeDef *huart, char c, int m)
{
 HAL_UART_Transmit(huart, (uint8_t*)&c, 1, HAL_MAX_DELAY);
 if(m == 1) HAL_UART_Transmit(huart, (uint8_t*)"\n\r", 2, HAL_MAX_DELAY);
}

//
// This function displays a new-line
//
void UART_SEND_NL(UART_HandleTypeDef *huart)
{
 HAL_UART_Transmit(huart, (uint8_t*)"\n\r", 2, HAL_MAX_DELAY);
}

//
// This function displays an integer number
//
void UART_SEND_INT(UART_HandleTypeDef *huart, int i, int m)
{
 char buffer[10];
 itoa(i, buffer, 10);
 HAL_UART_Transmit(huart, (uint8_t*) buffer, strlen(buffer), HAL_MAX_DELAY);
 if(m == 1) HAL_UART_Transmit(huart, (uint8_t*)"\n\r", 2, HAL_MAX_DELAY);
}

//
// Read an integer number
//
int ReadInt(void)
{
 int n, N = 0;
 char readBuf[1];
 while(1)
 {
 HAL_UART_Receive(&huart2, (uint8_t*)readBuf, 1, HAL_MAX_DELAY);
 UART_SEND_CHR(&huart2, readBuf[0], 0);
 if(readBuf[0] == '\r')break;
 n = atoi(readBuf);
 N = 10*N + n;
 }
 return (N);
}
```

```

//
// Start of main program. Read an integer number from the keyboard, then
// display the times table for that number
//
int main(void)
{
 int N, r1, r2;

 HAL_Init();

 SystemClock_Config();

 MX_GPIO_Init();
 MX_USART2_UART_Init();

 while (1)
 {
 UART_SEND_NL(&huart2);
 UART_SEND_TXT(&huart2, «LEARNING MULTIPLICATION», 1);
 UART_SEND_TXT(&huart2, «-----», 1);
 UART_SEND_NL(&huart2);

 r1 = rand() % 100;
 r2 = rand() % 100;
 UART_SEND_INT(&huart2, r1, 0);
 UART_SEND_TXT(&huart2, « X », 0);
 UART_SEND_INT(&huart2, r2, 0);
 UART_SEND_TXT(&huart2, « = », 0);

 N = ReadInt();
 UART_SEND_NL(&huart2);

 if(N == r1*r2)
 {
 UART_SEND_TXT(&huart2, «Well Done!», 1);
 }
 else
 {
 UART_SEND_TXT(&huart2, «Wrong!..Correct answer = », 0);
 UART_SEND_INT(&huart2, r1*r2, 1);
 }
 }
}

```

*Figure 11.20: Program listing.*

## 11.8 Project 5: Displaying Ambient Temperature on the PC Screen

### Description

In this project an analogue temperature sensor chip is connected to the development board. The ambient temperature is measured every second and sent to the PC through the UART where it is displayed.

### The aim

The aim of this project is to show how the temperature readings can be sent to the PC over the UART interface

### Block diagram

Figure 11.21 shows the block diagram of the project.



Figure 11.21: Block diagram of the project.

### Circuit diagram

The circuit diagram is shown in figure 11.22. A TMP36 type analogue temperature sensor chip is connected to analogue input PC2 of the development board.

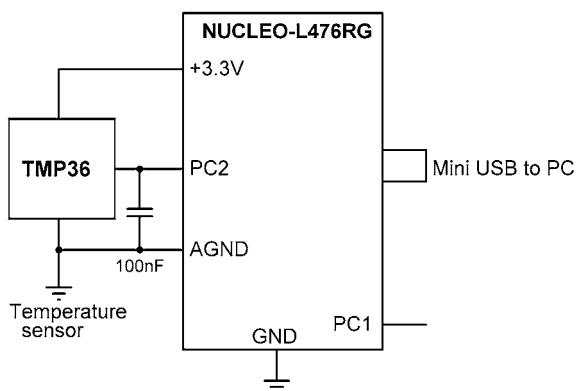


Figure 11.22: Circuit diagram of the project.

### Program listing

The steps are as follows.

- Start STM32CubeIDE as before.
- Create a new workspace.
- Select the board type as Nucleo-L476RG.
- Give the name **TEMPUART** to the project.

- Configure PC2 as ADC1\_IN3.
- Click the **Analog** tab and select **ADC1** and enable channel **IN3** as **Single-ended** to enable ADC on this channel (Pin PC2).
- Under **Parameter Settings**, click on **Resolution** and select **ADC 12-bit resolution**. Also, enable the **Continuous Conversion Mode**.
- Set **End of Conversion Selection** to **End of single conversions**.
- Set **Overrun behaviour** to **Overrun data overwritten**.
- Click **Clock Configuration** and set clock to 80 MHz.
- Select **USART2** under tab **Connectivity** and set its **Mode** to **Asynchronous**
- In **Parameter Settings**, set the **Baud Rate** to 9600, **Word Length** to 8, **Parity** to None, and **Stop Bits** to 1.
- Click **File**, followed by **Save** and click **YES** to generate code.
- Click **Core**, followed by **Src** and double click **main.c** to open the main program.

Inside the program loop the ADC is started, analogue data is read and stored in variable **adcResult**. This data is then converted into millivolts by multiplying with 3300 and dividing with 4095. This is because we have a 12-bit ADC and the reference voltage is 3.3V. i.e. 0 corresponds to 0 V, while 4095 corresponds to 3.3 V. The temperature is then calculated as a floating point number by subtracting 500 and dividing by 10. The temperature reading is then sent to the PC by calling function **UART\_SEND\_FLT**.

The program loop contains the following statements:

```
while (1)
{
 HAL_ADC_PollForConversion(&hadc1, 100); // wait for conv
 adcResult = HAL_ADC_GetValue(&hadc1); // Read temperature
 Temperature = ((float)adcResult) * 3300.0 / 4095.0;
 Temperature = (Temperature - 500.0) / 10.0; // To degrees C
 UART_SEND_TXT(&huart2, "Temperature = ", 0); // Send to PC
 UART_SEND_FLT(&huart2, Temperature, 1); // Send to PC
 HAL_Delay(1000); // Wait 1 sec
}
```

Compile the program in **Release** mode making sure there are no errors. Drag and drop the binary file **TEMPUART.bin** to device NUCLEO\_L476RG.

Figure 11.23 shows an output from the program, displayed on the terminal emulator screen.

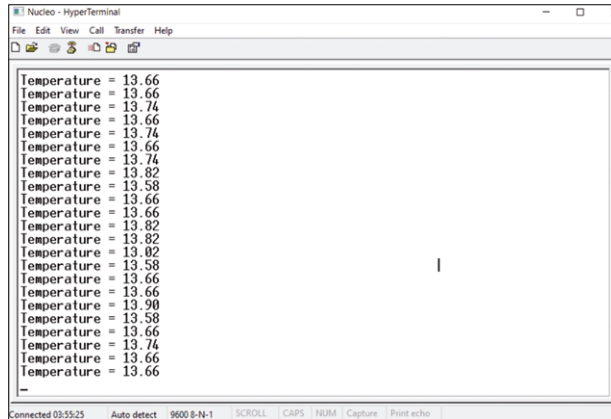


Figure 11.23: Output from the program.

Figure 11.24 shows the program listing.

```

/* USER CODE BEGIN Header */
/**

 * @file : main.c
 * @brief : Main program body

 * @attention
 *
 * <h2><center>© Copyright (c) 2020 STMicroelectronics.
 * All rights reserved.</center></h2>
 *
 * This software component is licensed by ST under BSD 3-Clause license,
 * the «License»; You may not use this file except in compliance with the
 * License. You may obtain a copy of the License at:
 *
 * opensource.org/licenses/BSD-3-Clause
 *

 */
#include «main.h»
#include «stdio.h»
#include «stdlib.h»
#include «string.h»

ADC_HandleTypeDef hadc1;

UART_HandleTypeDef huart2;

void SystemClock_Config(void);
static void MX_GPIO_Init(void);

```

```

static void MX_ADC1_Init(void);
static void MX_USART2_UART_Init(void);

void UART_SEND_TXT(UART_HandleTypeDef *huart, char buffer[], int m)
{
 HAL_UART_Transmit(huart, (uint8_t*) buffer, strlen(buffer), HAL_MAX_DELAY);
 if(m == 1) HAL_UART_Transmit(huart, (uint8_t*)»\n\r», 2, HAL_MAX_DELAY);
}

void UART_SEND_FLT(UART_HandleTypeDef *huart, float p, int m)
{
 char buffer[10];
 sprintf(buffer, «%5.2f», p);
 HAL_UART_Transmit(huart, (uint8_t*) buffer, strlen(buffer), HAL_MAX_DELAY);
 if(m == 1) HAL_UART_Transmit(huart, (uint8_t*)»\n\r», 2, HAL_MAX_DELAY);
}

//
// Start of main program
//
int main(void)
{
 uint32_t adcResult;
 float Temperature;
 HAL_Init();

 SystemClock_Config();

 MX_GPIO_Init();
 MX_ADC1_Init();
 MX_USART2_UART_Init();

 HAL_ADC_Start(&hadc1);

 while (1)
 {
 HAL_ADC_PollForConversion(&hadc1, 100); // wait for conv
 adcResult = HAL_ADC_GetValue(&hadc1); // Read temperature
 Temperature = ((float)adcResult) * 3300.0 / 4095.0; // To mV
 Temperature = (Temperature - 500.0) / 10.0; // To degrees C
 UART_SEND_TXT(&huart2, «Temperature = », 0); // Send to PC
 UART_SEND_FLT(&huart2, Temperature, 1); // Send to PC
 HAL_Delay(1000); // Wait 1 sec
 }
}

```



```
void SystemClock_Config(void)
{
 RCC_OscInitTypeDef RCC_OscInitStruct = {0};
 RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};
 RCC_PeriphCLKInitTypeDef PeriphClkInit = {0};

 RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
 RCC_OscInitStruct.HSIState = RCC_HSI_ON;
 RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
 RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
 RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
 RCC_OscInitStruct.PLL.PLLM = 2;
 RCC_OscInitStruct.PLL.PLLN = 20;
 RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV7;
 RCC_OscInitStruct.PLL.PLLQ = RCC_PLLQ_DIV2;
 RCC_OscInitStruct.PLL.PLLR = RCC_PLLR_DIV2;
 if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
 {
 Error_Handler();
 }

 RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
 |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
 RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
 RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
 RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
 RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

 if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_4) != HAL_OK)
 {
 Error_Handler();
 }
 PeriphClkInit.PeriphClockSelection = RCC_PERIPHCLK_USART2|RCC_PERIPHCLK_ADC;
 PeriphClkInit.Usart2ClockSelection = RCC_USART2CLKSOURCE_PCLK1;
 PeriphClkInit.AdcClockSelection = RCC_ADCCLKSOURCE_PLLSAI1;
 PeriphClkInit.PLLSAI1.PLLSAI1Source = RCC_PLLSOURCE_HSI;
 PeriphClkInit.PLLSAI1.PLLSAI1M = 2;
 PeriphClkInit.PLLSAI1.PLLSAI1N = 8;
 PeriphClkInit.PLLSAI1.PLLSAI1P = RCC_PLLP_DIV7;
 PeriphClkInit.PLLSAI1.PLLSAI1Q = RCC_PLLQ_DIV2;
 PeriphClkInit.PLLSAI1.PLLSAI1R = RCC_PLLR_DIV2;
 PeriphClkInit.PLLSAI1.PLLSAI1ClockOut = RCC_PLLSAI1_ADC1CLK;
 if (HAL_RCCEx_PeriphCLKConfig(&PeriphClkInit) != HAL_OK)
 {
 Error_Handler();
 }
}
```

```

 if (HAL_PWREx_ControlVoltageScaling(PWR_REGULATOR_VOLTAGE_SCALE1) != HAL_OK)
 {
 Error_Handler();
 }
}

static void MX_ADC1_Init(void)
{
 ADC_MultiModeTypeDef multimode = {0};
 ADC_ChannelConfTypeDef sConfig = {0};

 hadc1.Instance = ADC1;
 hadc1.Init.ClockPrescaler = ADC_CLOCK_ASYNC_DIV1;
 hadc1.Init.Resolution = ADC_RESOLUTION_12B;
 hadc1.Init.DataAlign = ADC_DATAALIGN_RIGHT;
 hadc1.Init.ScanConvMode = ADC_SCAN_DISABLE;
 hadc1.Init.EOCSelection = ADC_EOC_SINGLE_CONV;
 hadc1.Init.LowPowerAutoWait = DISABLE;
 hadc1.Init.ContinuousConvMode = ENABLE;
 hadc1.Init.NbrOfConversion = 1;
 hadc1.Init.DiscontinuousConvMode = DISABLE;
 hadc1.Init.NbrOfDiscConversion = 1;
 hadc1.Init.ExternalTrigConv = ADC_SOFTWARE_START;
 hadc1.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_NONE;
 hadc1.Init.DMAContinuousRequests = DISABLE;
 hadc1.Init.Overrun = ADC_OVR_DATA_OVERWRITTEN;
 hadc1.Init.OversamplingMode = DISABLE;
 if (HAL_ADC_Init(&hadc1) != HAL_OK)
 {
 Error_Handler();
 }

 multimode.Mode = ADC_MODE_INDEPENDENT;
 if (HAL_ADCEx_MultiModeConfigChannel(&hadc1, &multimode) != HAL_OK)
 {
 Error_Handler();
 }

 sConfig.Channel = ADC_CHANNEL_3;
 sConfig.Rank = ADC_REGULAR_RANK_1;
 sConfig.SamplingTime = ADC_SAMPLETIME_2CYCLES_5;
 sConfig.SingleDiff = ADC_SINGLE_ENDED;
 sConfig.OffsetNumber = ADC_OFFSET_NONE;
 sConfig.Offset = 0;
 if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)

```

```
{
 Error_Handler();
}

static void MX_USART2_UART_Init(void)
{
 huart2.Instance = USART2;
 huart2.Init.BaudRate = 9600;
 huart2.Init.WordLength = UART_WORDLENGTH_8B;
 huart2.Init.StopBits = UART_STOPBITS_1;
 huart2.Init.Parity = UART_PARITY_NONE;
 huart2.Init.Mode = UART_MODE_TX_RX;
 huart2.Init.HwFlowCtl = UART_HWCONTROL_NONE;
 huart2.Init.OverSampling = UART_OVERSAMPLING_16;
 huart2.Init.OneBitSampling = UART_ONE_BIT_SAMPLE_DISABLE;
 huart2.AdvancedInit.AdvFeatureInit = UART_ADVFEATURE_NO_INIT;
 if (HAL_UART_Init(&huart2) != HAL_OK)
 {
 Error_Handler();
 }
}

static void MX_GPIO_Init(void)
{
 /* GPIO Ports Clock Enable */
 __HAL_RCC_GPIOC_CLK_ENABLE();
 __HAL_RCC_GPIOA_CLK_ENABLE();
}

void Error_Handler(void)
{
}

#ifdef USE_FULL_ASSERT

void assert_failed(uint8_t *file, uint32_t line)
{
}

#endif

/***** (C) COPYRIGHT STMicroelectronics *****/
```

*Figure 11.24: The TEMPUART program.*

### 11.9 Project 6: Communicating with Arduino (Displaying Temperature)

#### Description

This project is about communication between an Arduino and the Nucleo-L476RG development board. An analogue temperature sensor chip is connected to one of the Arduino analogue input ports. Arduino reads the temperature every second and sends the result to the NucleoL476RG development board through UART. Another UART is used on the Nucleo-L476RG development board to send the temperature to the PC where it is displayed by the terminal emulator software.

#### Block diagram

Figure 11.25 shows the block diagram of the project.

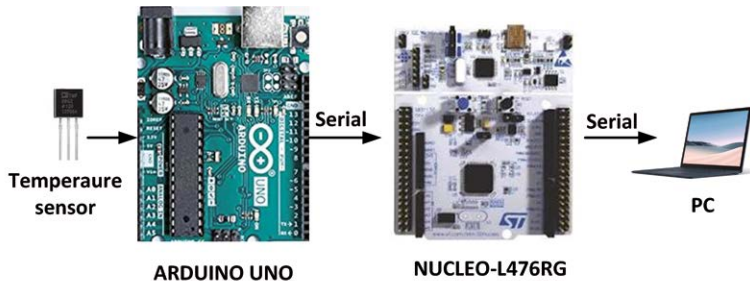


Figure 11.25: Block diagram of the project.

#### Circuit diagram

The circuit diagram of the project is shown in Figure 11.26. TMP36 temperature sensor chip is connected to analogue input A0 of the Arduino. Ports A6 and A7 of Arduino are configured as serial software RX and TX pins and are connected to pins PA0 (Serial4 TX) and PA1 (Serial4 RX) pins of the Nucleo board. Notice that only the serial output is used from the Arduino.

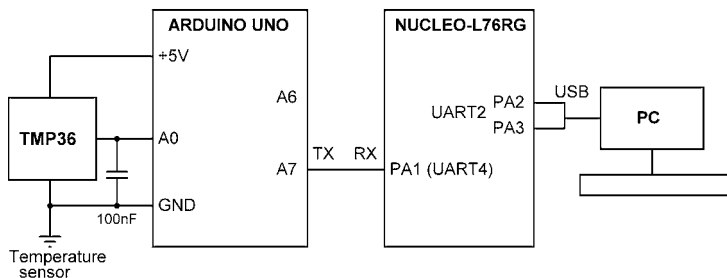


Figure 11.26: Circuit diagram of the project.

## Program listing

### Nucleo-L476RG software configuration

The steps are as follows.

- Start STM32CubeIDE as before.
- Create a new workspace.
- Select the board type as Nucleo-L476RG.
- Give the name **ARDUINO** to the project.
- Click **Clock Configuration** and set clock to 80 MHz.
- Select **USART2** under tab **Connectivity** and set its **Mode** to **Asynchronous**
- In **Parameter Settings**, set the **Baud Rate** to 9600, **Word Length** to 8, **Parity** to None, and **Stop Bits** to 1.
- Select **UART4** under tab **Connectivity** and set its **Mode** to **Asynchronous**
- In **Parameter Settings**, set the **Baud Rate** to 9600, **Word Length** to 8, **Parity** to None, and **Stop Bits** to 1.
- Click **File**, followed by **Save** and click **YES** to generate code.
- Click **Core**, followed by **Src** and double click **main.c** to open the main program.

### Arduino program

The Arduino program (program: **ArdTemp.c**) listing is shown in Figure 11.27. At the beginning of the program, the **SoftwareSerial** library is included. GPIO pins 6 and 7 are then configured as RX and TX pins of software serial port respectively. Inside the setup routine the baud rate of the serial port is set to 9600. The main program loop reads the analogue data from the sensor and converts it to degrees centigrade by subtracting 500.0 and dividing by 10.0. The temperature reading is then stored as an integer in variable **temp** for simplicity, and is sent out through the serial port by calling function **MySerial.print**. The temperature data is sent as two characters with the MSB byte sent first, followed by the LSB byte. If the temperature is less than 10, then 0 is added to make its size 2 bytes.

```

/*****
 *
 * TEMPERATURE SENSOR
 *
 * =====
 *
 * This program reads the analog temperature and sends
 * it to the Nucleo-L476RG board over serial UART
 *
 * Author: Dogan Ibrahim
 * Date : October, 2020
 * File : ArdTemp.c
 *****/
#include <SoftwareSerial.h>
SoftwareSerial MySerial(6, 7); // RX, TX
int TempPin = 0;
int val, temp;
float mv, temperature;
String tempstr;

```

```

void setup()
{
 MySerial.begin(9600);
}

void loop()
{
 val = analogRead(TempPin);
 mv = val * 5000.0 / 1024.0;
 temperature = (mv - 500.0) / 10.0;
 temp = (int)temperature;

 if(temp < 10)
 tempstr = «0» + String(temp);
 else
 tempstr = String(temp);

 MySerial.print(tempstr);
 delay(1000);
}

```

Figure 11.27: The Arduino program.

### Nucleo-L476RG program

Figure 11.28 shows the Nucleo program (program name: **ARDUINO**). The following user functions are used in this program:

```

void UART_SEND_TXT(UART_HandleTypeDef *huart, char buffer[], int m)
void UART_SEND_INT(UART_HandleTypeDef *huart, int i, int m)

```

Inside the program loop the program displays heading **Arduino Temperature =** and then waits to 2 receives 2 bytes from the Arduino. Notice that data is received by **UART4** as follows:

```

HAL_UART_Receive(&huart4, (uint8_t*)readBuf, 2, HAL_MAX_DELAY);

```

The received bytes are stored in character array **readBuf**. The temperature reading is then extracted by multiplying the first byte by 10 and adding to the second byte after converting them to numeric data. Function **UART\_SEND\_INT** sends the reading to **UART2** which displays the data on the PC terminal emulator screen.

```

/* USER CODE BEGIN Header */
/**
 * *****
 * @file : main.c
 * @brief : Main program body
 * *****
 */

```

```
* @attention
*
* <h2><center>© Copyright (c) 2020 STMicroelectronics.
* All rights reserved.</center></h2>
*
* This software component is licensed by ST under BSD 3-Clause license,
* the «License»; You may not use this file except in compliance with the
* License. You may obtain a copy of the License at:
*
* opensource.org/licenses/BSD-3-Clause
*

*/
#include «main.h»
#include «stdio.h»
#include «stdlib.h»
#include «string.h»

UART_HandleTypeDef huart4;
UART_HandleTypeDef huart2;

void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_UART4_Init(void);
static void MX_USART2_UART_Init(void);

//
// This function displays text
//
void UART_SEND_TXT(UART_HandleTypeDef *huart, char buffer[], int m)
{
 HAL_UART_Transmit(huart, (uint8_t*) buffer, strlen(buffer), HAL_MAX_DELAY);
 if(m == 1) HAL_UART_Transmit(huart, (uint8_t*)»\n\r», 2, HAL_MAX_DELAY);
}

//
// This function displays an integer number
//
void UART_SEND_INT(UART_HandleTypeDef *huart, int i, int m)
{
 char buffer[10];
 itoa(i, buffer, 10);
 HAL_UART_Transmit(huart, (uint8_t*) buffer, strlen(buffer), HAL_MAX_DELAY);
 if(m == 1) HAL_UART_Transmit(huart, (uint8_t*)»\n\r», 2, HAL_MAX_DELAY);
}
```

```

int main(void)
{
 char readBuf[2];
 int T;

 HAL_Init();

 SystemClock_Config();

 MX_GPIO_Init();
 MX_UART4_Init();
 MX_USART2_UART_Init();

 while (1)
 {
 UART_SEND_TXT(&huart2, «Arduino Temperature = », 0);
 HAL_UART_Receive(&huart4, (uint8_t*)readBuf, 2, HAL_MAX_DELAY);
 T = 10*(readBuf[0]-'0') + readBuf[1]-'0';
 UART_SEND_INT(&huart2, T, 1);
 }
}

void SystemClock_Config(void)
{
 RCC_OscInitTypeDef RCC_OscInitStruct = {0};
 RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};
 RCC_PeriphCLKInitTypeDef PeriphClkInit = {0};

 RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
 RCC_OscInitStruct.HSISState = RCC_HSI_ON;
 RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
 RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
 RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
 RCC_OscInitStruct.PLL.PLLM = 2;
 RCC_OscInitStruct.PLL.PLLN = 20;
 RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV7;
 RCC_OscInitStruct.PLL.PLLQ = RCC_PLLQ_DIV2;
 RCC_OscInitStruct.PLL.PLLR = RCC_PLLR_DIV2;
 if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
 {
 Error_Handler();
 }

 RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
 |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;

```



```
RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_4) != HAL_OK)
{
 Error_Handler();
}
PeriphClkInit.PeriphClockSelection = RCC_PERIPHCLK_USART2|RCC_PERIPHCLK_UART4;
PeriphClkInit.Usart2ClockSelection = RCC_USART2CLKSOURCE_PCLK1;
PeriphClkInit.Uart4ClockSelection = RCC_UART4CLKSOURCE_PCLK1;
if (HAL_RCCEx_PeriphCLKConfig(&PeriphClkInit) != HAL_OK)
{
 Error_Handler();
}

if (HAL_PWREx_ControlVoltageScaling(PWR_REGULATOR_VOLTAGE_SCALE1) != HAL_OK)
{
 Error_Handler();
}
}

static void MX_UART4_Init(void)
{
 huart4.Instance = UART4;
 huart4.Init.BaudRate = 9600;
 huart4.Init.WordLength = UART_WORDLENGTH_8B;
 huart4.Init.StopBits = UART_STOPBITS_1;
 huart4.Init.Parity = UART_PARITY_NONE;
 huart4.Init.Mode = UART_MODE_TX_RX;
 huart4.Init.HwFlowCtl = UART_HWCONTROL_NONE;
 huart4.Init.OverSampling = UART_OVERSAMPLING_16;
 huart4.Init.OneBitSampling = UART_ONE_BIT_SAMPLE_DISABLE;
 huart4.AdvancedInit.AdvFeatureInit = UART_ADVFEATURE_NO_INIT;
 if (HAL_UART_Init(&huart4) != HAL_OK)
 {
 Error_Handler();
 }
}

static void MX_USART2_UART_Init(void)
{
 huart2.Instance = USART2;
 huart2.Init.BaudRate = 9600;
 huart2.Init.WordLength = UART_WORDLENGTH_8B;
```

```

 huart2.Init.StopBits = UART_STOPBITS_1;
 huart2.Init.Parity = UART_PARITY_NONE;
 huart2.Init.Mode = UART_MODE_TX_RX;
 huart2.Init.HwFlowCtl = UART_HWCONTROL_NONE;
 huart2.Init.OverSampling = UART_OVERSAMPLING_16;
 huart2.Init.OneBitSampling = UART_ONE_BIT_SAMPLE_DISABLE;
 huart2.AdvancedInit.AdvFeatureInit = UART_ADVFEATURE_NO_INIT;
 if (HAL_UART_Init(&huart2) != HAL_OK)
 {
 Error_Handler();
 }
}

static void MX_GPIO_Init(void)
{
 /* GPIO Ports Clock Enable */
 __HAL_RCC_GPIOA_CLK_ENABLE();
}

void Error_Handler(void)
{
}

#ifdef USE_FULL_ASSERT

void assert_failed(uint8_t *file, uint32_t line)
{
}

#endif

/***** (C) COPYRIGHT STMicroelectronics *****/

```

*Figure 11.28: The ARDUINO program.*

An example output from the program is shown in Figure 11.29.

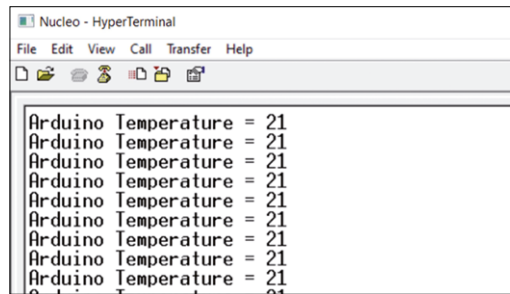


Figure 11.29: Example output from the program.

### 11.10 UART in interrupt mode

The program in the last project was based on UART polling mode where the program had to block until data is received by the UART. This kind of behaviour may not be accessible in many time critical real-time systems where the CPU may be required to do other tasks. To address this issue, HAL offers interrupt based UART operations where an interrupt is generated each time data is sent or received by the UART.

Function **HAL\_UART\_Transmit\_IT** is used to transmit a sequence of bytes in interrupt mode. Similarly, function and **HAL\_UART\_Receive\_IT** is used to receive a sequence of bytes in UART interrupt mode without blocking.

The STM32 MCUs offer several UART based interrupts. A list of some important UART based interrupt sources are given below. These sources will generate interrupts if their corresponding interrupt enable bits are set:

- Transmit data register empty
- Transmission complete
- Received data ready to be read
- Overrun error
- Parity error
- Break detection

An example project is given in the next section which uses UART in interrupt mode to receive data from the Arduino.

### 11.11 Project 7: Communicating with Arduino – UART Interrupt Mode

#### Description

This project is similar to the previous one where the Arduino Uno sends the ambient temperature readings to the Nucleo-L476RG development board. In this project the UART receives data inside the interrupt service routine.

#### Block diagram

The block diagram of the project is same as in Figure 11.25.

#### Circuit diagram

The circuit diagram of the project is same as in Figure 11.26.

**Arduino program:** The Arduino program is same as the one given in Figure 11.27 where the temperature readings are sent every second through a software based serial port.

### Nucleo-L476RG software configuration

The steps are given below.

- Start STM32CubeIDE as before.
- Create a new workspace.
- Select the board type as Nucleo-L476RG.
- Give the name **UARTINT** to the project.
- Click **Clock Configuration** and set clock to 80 MHz.
- Select **USART2** under tab **Connectivity** and set its **Mode** to **Asynchronous**
- In **Parameter Settings**, set the **Baud Rate** to 9600, **Word Length** to 8, **Parity** to None, and **Stop Bits** to 1.
- Select **UART4** under tab **Connectivity** and set its **Mode** to **Asynchronous**.
- In **Parameter Settings**, set the **Baud Rate** to 9600, **Word Length** to 8, **Parity** to None, and **Stop Bits** to 1.
- Click the 'NVIC Settings' tab and enable UART4 global interrupt (Figure 11.30).

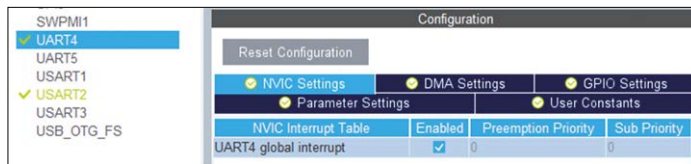


Figure 11.30: Configuring NVIC Settings.

- Click **File**, followed by **Save** and click **YES** to generate code.
- Click **Core**, followed by **Src** and double click **main.c** to open the main program.

The following user functions are used in the program:

```
oid UART_SEND_TXT(UART_HandleTypeDef *huart, char buffer[], int m)
void UART_SEND_INT(UART_HandleTypeDef *huart, int i, int m)
```

Enter the following statement before the program loop to start the UART interrupts. Notice that UART4 is used in this statement since we want to receive serial data from the Arduino. The number of bytes we want to receive is set to 2:

```
HAL_UART_Receive_IT(&huart4, (uint8_t *)readBuf, 2);
```

The following callback interrupt service routine function is called whenever the specified number of bytes are received by the UART:

```
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart) is called
```

Inside this function we send the heading **Ambient Temperature =** to the PC (through UART2), extract the temperature and store in variable **T**. The character array **readBuf**

stores the two bytes received from the Arduino (these bytes are as ASCII characters, e.g. number 2 is read as 0x32). This value is then sent to the PC. Notice that the UART interrupts must be re-started by entering the following statement inside the callback function:

```
HAL_UART_Receive_IT(&huart4, (uint8_t *)readBuf, 2);
```

The code inside the callback interrupt service routine function is as follows:

```
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
{
 int T;
 UART_SEND_TXT(&huart2, "Ambient Temperature = ", 0);
 T = 10*(readBuf[0]-'0') + readBuf[1]-'0';
 UART_SEND_INT(&huart2, T, 1);
 HAL_UART_Receive_IT(&huart4, (uint8_t *)readBuf, 2);
}
```

Figure 11.31 shows the program listing (comments have been removed for clarity).

```
/* USER CODE BEGIN Header */
/**
 * *****
 * @file : main.c
 * @brief : Main program body
 * *****
 * @attention
 *
 * <h2><center>© Copyright (c) 2020 STMicroelectronics.
 * All rights reserved.</center></h2>
 *
 * This software component is licensed by ST under BSD 3-Clause license,
 * the «License»; You may not use this file except in compliance with the
 * License. You may obtain a copy of the License at:
 *
 * opensource.org/licenses/BSD-3-Clause
 *
 * *****
 */
#include «main.h»
#include «stdio.h»
#include «stdlib.h»
#include «string.h»

UART_HandleTypeDef huart4;
UART_HandleTypeDef huart2;

void SystemClock_Config(void);
```

```

static void MX_GPIO_Init(void);
static void MX_UART4_Init(void);
static void MX_USART2_UART_Init(void);

char readBuf[2];

//
// This function displays text
//
void UART_SEND_TXT(UART_HandleTypeDef *huart, char buffer[], int m)
{
 HAL_UART_Transmit(huart, (uint8_t*) buffer, strlen(buffer), HAL_MAX_DELAY);
 if(m == 1) HAL_UART_Transmit(huart, (uint8_t*)»\n\r», 2, HAL_MAX_DELAY);
}

//
// This function displays an integer number
//
void UART_SEND_INT(UART_HandleTypeDef *huart, int i, int m)
{
 char buffer[10];
 itoa(i, buffer, 10);
 HAL_UART_Transmit(huart, (uint8_t*) buffer, strlen(buffer), HAL_MAX_DELAY);
 if(m == 1) HAL_UART_Transmit(huart, (uint8_t*)»\n\r», 2, HAL_MAX_DELAY);
}

//
// This is the UART interrupt callback function
//
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
{
 int T;
 UART_SEND_TXT(&huart2, «Ambient Temperature = », 0); // Send heading
 T = 10*(readBuf[0]-’0’) + readBuf[1]-’0’; // Temp in T
 UART_SEND_INT(&huart2, T, 1); // Send to UART2
 HAL_UART_Receive_IT(&huart4, (uint8_t *)readBuf, 2); // Restart rcv int
}

//
// Start of main program
//
int main(void)
{
 char readBuf[2];
 HAL_Init();

```

```
SystemClock_Config();

MX_GPIO_Init();
MX_UART4_Init();
MX_USART2_UART_Init();

HAL_UART_Receive_IT(&huart4, (uint8_t *)readBuf, 2); // Start the rcv int

while (1)
{
}

}

void SystemClock_Config(void)
{
 RCC_OscInitTypeDef RCC_OscInitStruct = {0};
 RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};
 RCC_PeriphCLKInitTypeDef PeriphClkInit = {0};

 RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
 RCC_OscInitStruct.HSISState = RCC_HSI_ON;
 RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
 RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
 RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
 RCC_OscInitStruct.PLL.PLLM = 2;
 RCC_OscInitStruct.PLL.PLLN = 20;
 RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV7;
 RCC_OscInitStruct.PLL.PLLQ = RCC_PLLQ_DIV2;
 RCC_OscInitStruct.PLL.PLLR = RCC_PLLR_DIV2;
 if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
 {
 Error_Handler();
 }

 RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
 |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
 RCC_ClkInitStruct.SYSClkSource = RCC_SYSCCLKSOURCE_PLLCLK;
 RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCCLK_DIV1;
 RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
 RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

 if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_4) != HAL_OK)
 {
 Error_Handler();
 }
 PeriphClkInit.PeriphClockSelection = RCC_PERIPHCLK_USART2|RCC_PERIPHCLK_UART4;
```

```

PeriphClkInit.Usart2ClockSelection = RCC_USART2CLKSOURCE_PCLK1;
PeriphClkInit.Uart4ClockSelection = RCC_UART4CLKSOURCE_PCLK1;
if (HAL_RCCEx_PeriphCLKConfig(&PeriphClkInit) != HAL_OK)
{
 Error_Handler();
}

if (HAL_PWREx_ControlVoltageScaling(PWR_REGULATOR_VOLTAGE_SCALE1) != HAL_OK)
{
 Error_Handler();
}
}

static void MX_UART4_Init(void)
{
 huart4.Instance = UART4;
 huart4.Init.BaudRate = 9600;
 huart4.Init.WordLength = UART_WORDLENGTH_8B;
 huart4.Init.StopBits = UART_STOPBITS_1;
 huart4.Init.Parity = UART_PARITY_NONE;
 huart4.Init.Mode = UART_MODE_TX_RX;
 huart4.Init.HwFlowCtl = UART_HWCONTROL_NONE;
 huart4.Init.OverSampling = UART_OVERSAMPLING_16;
 huart4.Init.OneBitSampling = UART_ONE_BIT_SAMPLE_DISABLE;
 huart4.AdvancedInit.AdvFeatureInit = UART_ADVFEATURE_NO_INIT;
 if (HAL_UART_Init(&huart4) != HAL_OK)
 {
 Error_Handler();
 }
}

static void MX_USART2_UART_Init(void)
{
 huart2.Instance = USART2;
 huart2.Init.BaudRate = 9600;
 huart2.Init.WordLength = UART_WORDLENGTH_8B;
 huart2.Init.StopBits = UART_STOPBITS_1;
 huart2.Init.Parity = UART_PARITY_NONE;
 huart2.Init.Mode = UART_MODE_TX_RX;
 huart2.Init.HwFlowCtl = UART_HWCONTROL_NONE;
 huart2.Init.OverSampling = UART_OVERSAMPLING_16;
 huart2.Init.OneBitSampling = UART_ONE_BIT_SAMPLE_DISABLE;
 huart2.AdvancedInit.AdvFeatureInit = UART_ADVFEATURE_NO_INIT;
 if (HAL_UART_Init(&huart2) != HAL_OK)
 {

```



```
 Error_Handler();
}
}

static void MX_GPIO_Init(void)
{
 /* GPIO Ports Clock Enable */
 __HAL_RCC_GPIOA_CLK_ENABLE();
}

void Error_Handler(void)
{

}

#ifdef USE_FULL_ASSERT

void assert_failed(uint8_t *file, uint32_t line)
{

}

#endif

/***** (C) COPYRIGHT STMicroelectronics *****/
```

*Figure 11.31: The UARTINT program.*

### 11.12 Using UART in DMA mode

It is also possible to program the Nucleo-L476RG development board so that the UART operates in DMA mode. The advantage of this mode is that the data transfer takes place in non-blocking mode and without intervention of the CPU.

The programming for the DMA mode is very similar to programming in interrupt mode. The DMA for UART4 must be set for this mode in UART4 configuration. In DMA mode, when half of the data is received, the callback function **HAL\_UART\_RxHalfCpltCallback** is called. Also, when the complete data is received, the callback function **HAL\_UART\_RxCpltCallback** is called. User code can be inserted inside these functions. The following function must also be called before the program loop and also inside the data complete callback function (for UART4 receiving two bytes):

```
HAL_UART_Receive_DMA (&huart4, (uint8_t*)readBuf, 2);
```

### 11.13 Summary

In this Chapter we have learned how to use the UART in several projects for serial communication between two devices. In the next Chapter we will be concentrating on the topic of the I2C bus and develop several projects based on using it.

## CHAPTER 12 • The I<sup>2</sup>C Bus Interface

### 12.1 Overview

The I<sup>2</sup>C bus (also sloppily written 'I2C bus' or hypercorrectly 'Inter-Integrated-Circuit Bus') is commonly used in microcontroller-based projects. This is a hardware specification and protocol developed by the semiconductor division of Philips back in 1982.

In this Chapter we shall be looking at the use of this bus with our Nucleo-L476RG dev board. The STM32L476RG processor includes three I<sup>2</sup>C bus interfaces. The aim is to make the reader familiar with the I<sup>2</sup>C bus library functions and to show how they can be used in a real project. Before looking at the details of the project it is worthwhile to look at the basic principles of the I<sup>2</sup>C bus.

### 12.2 The I<sup>2</sup>C Bus

I<sup>2</sup>C bus is one of the most used microcontroller communication protocols for communicating with external devices such as sensors and actuators. The I<sup>2</sup>C bus is a single master, multiple slave, half-duplex, single-ended, 8-bit bus and it can operate at standard mode: 100 Kbit/s, full speed: 400 Kbit/s, fast mode: 1 Mbit/s, and high speed: 3.2 Mbit/s. The bus consists of two open-drain wires, pulled-up with resistors. The sizing of these resistors is directly connected with the bus capacitance and the transmission speed:

**SDA:** data line

**SCL:** clock line

Figure 12.1 shows the structure of an I<sup>2</sup>C bus with one master and three slaves.

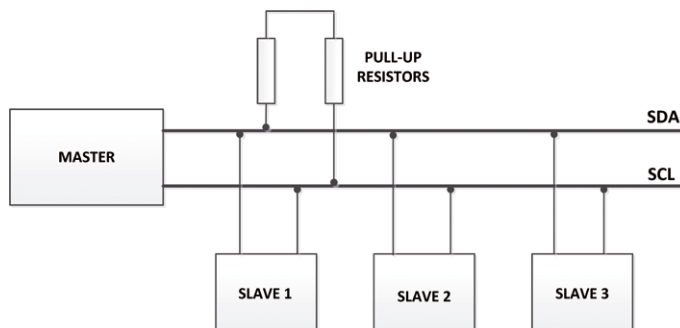


Figure 12.1: I<sup>2</sup>C bus with one master and three slaves.

Because the I<sup>2</sup>C bus is based on just two wires, there should be a way to address an individual slave device on the same bus. For this reason, the protocol defines that each slave device provides a unique slave address for the given bus. This address is usually 7-bits wide. When the bus is free both lines are HIGH. All communication on the bus is initiated and completed by the master which initially sends a START bit, and completes a transaction by sending STOP bit. This alerts all the slaves that some data is coming on the bus and all the slaves listen on the bus. After the start bit, 7 bits of unique slave address is sent. Each slave device on the bus has its own address and this ensures that only the addressed slave communicates on the bus at any time to avoid any collisions. The last sent bit is read/write

bit such that if this bit is 0, it means that the master wishes to write to the bus (e.g. to a register of a slave), if this bit is a 1, it means that the master wishes to read from the bus (e.g. from the register of a slave). The data is sent on the bus with the MSB bit first. An acknowledgement (ACK) bit takes place after every byte and this bit allows the receiver to signal the transmitter that the byte was received successfully and as a result another byte may be sent. The ACK bit is sent at the 9th clock pulse.

The communication over the I<sup>2</sup>C bus is simply as follows:

- the master sends on the bus the address of the slave it wants to communicate with;
- the LSB is the R/W bit which establishes the direction of data transmission, i.e. from master to slave (R/W = 0), or from slave to master (R/W = 1);
- the required bytes are sent, each interleaved with an ACK bit, until a stop condition occurs.

Depending on the type of slave device used, some transactions may require separate transaction. For example, the steps to read data from an I<sup>2</sup>C compatible memory device are:

- the Master starts the transaction in write mode (R/W = 0) by sending the slave address on the bus;
- the memory location to be retrieved are then sent as two bytes (assuming 64 Kbit memory);
- the master sends a STOP condition to end the transaction;
- The master starts a new transaction in read mode (R/W = 1) by sending the slave address on the bus;
- the master reads the data from the memory. If reading the memory in sequential format, then more than one byte will be read;
- the master sets a stop condition on the bus.

Luckily the STM32CubeIDE software handles all the required conditions on the I<sup>2</sup>C bus, thus simplifying the task of developing I<sup>2</sup>C based projects.

### 12.3 STM32L476RG I<sup>2</sup>C ports

The STM32L476RG MCU has 3 built-in I<sup>2</sup>C modules, each one capable of operating up to 1MHz. These modules use the following GPIO pins:

| I <sup>2</sup> C module | GPIO SDA pin | GPIO SCL pin |
|-------------------------|--------------|--------------|
| I2C1                    | PB7          | PB6          |
| I2C2                    | PB11         | PB10         |
| I2C3                    | PC1          | PC0          |

As alternatives, we can also use the following pins:

|      |      |      |
|------|------|------|
| I2C1 | PB9  | PB8  |
| I2C2 | PB14 | PB13 |

## 12.4 Project 1: Port Expander

### Description

A simple project is given in this section to show how the I<sup>2</sup>C functions can be used in a program. In this project the I<sup>2</sup>C bus compatible Port Expander chip (MCP23017) is used to give additional 16 I/O ports to the Nucleo-L476RG board. This is useful in some applications where a large number of I/O ports may be required. In this project an LED is connected to MCP23017 port pin GPA0 (pin 21) and the LED is flashed ON and OFF every second. A 330-ohm current limiting resistor is used in series with the LED.

### The aim

The aim of this project is to show how the I<sup>2</sup>C bus can be used in Nucleo-L476RG based projects.

### Block diagram

The block diagram of the project is shown in Figure 12.2.

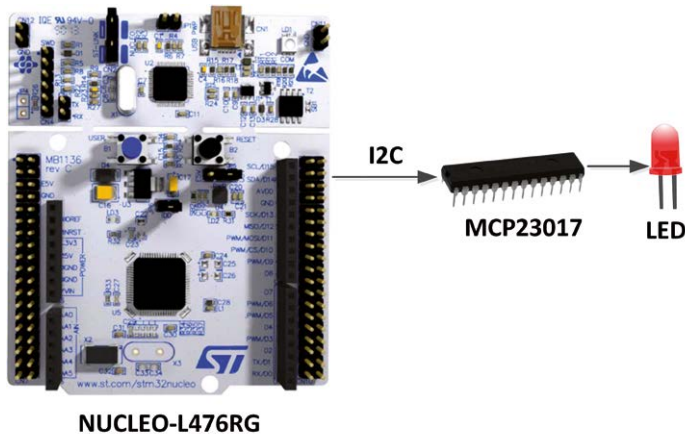


Figure 12.2: Block diagram of the project.

### The MCP23017

The MCP23017 is a 28-pin chip with the following features. The pin configuration is shown in Figure 12.3:

- 16 bi-directional I/O ports
- up to 1.7MHz operation on I<sup>2</sup>C bus
- interrupt capability
- external reset input
- low standby current
- +1.8 to +5.5V operation
- three address pins so that up to eight devices can be used on the I<sup>2</sup>C bus
- 28-pin DIL package

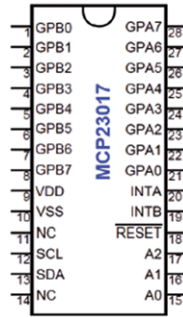


Figure 12.3: Pin configuration of the MCP23017.

The pin descriptions are given in Table 12.1.

| Pin       | Description      |
|-----------|------------------|
| GPA0-GPA7 | Port A pins      |
| GPB0-GPB7 | Port B pins      |
| VDD       | Power supply     |
| VSS       | Ground           |
| SDA       | I2C data pin     |
| SCL       | I2C clock pin    |
| RESET     | Reset pin        |
| A0-A2     | I2C address pins |

Table 12.1: MCP23017 pin descriptions

The MCP23017 is addressed using pins A0 to A2. Table 12.2 shows the address selection. In this project the address pins are connected to ground, thus the address of the chip is 0x20. The chip address is 7 bits wide with the low bit is set or cleared depending on whether we wish to read data from the chip or write data to the chip respectively. Since in this project we will be writing to the MCP23017, the low bit should be 0, making the chip byte address (also called the **device opcode**) as 0x40.

| A2 | A1 | A0 | Address |
|----|----|----|---------|
| 0  | 0  | 0  | 0x40    |
| 0  | 0  | 1  | 0x21    |
| 0  | 1  | 0  | 0x22    |
| 0  | 1  | 1  | 0x23    |
| 1  | 0  | 0  | 0x24    |
| 1  | 0  | 1  | 0x25    |
| 1  | 1  | 0  | 0x26    |
| 1  | 1  | 1  | 0x27    |

Table 12.2: Address selectin of the MCP23017.

The MCP23017 chip has 8 internal registers that can be configured for its operation. The device can either be operated in 16-bit mode or in two 8-bit mode by configuring bit IO-CON.BANK. On power-up this bit is cleared which chooses the two 8-bit mode by default. The I/O direction of the port pins are controlled with registers IODIRA (at address 0x00) and IODIRB (at address 0x01). Clearing a bit to 0 in these registers makes the corresponding port pin(s) as output(s). Similarly, setting a bit to 1 in these registers make the corresponding port pin(s) input(s). GPIOA and GPIOB register addresses are 0x12 and 0x13 respectively. This is shown in Figure 12.4.

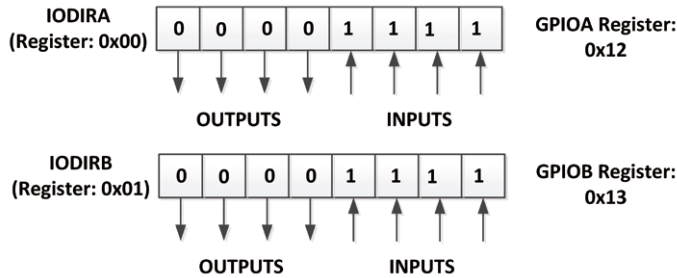


Figure 12.4: Configuring the I/O ports.

Figure 12.5 shows the circuit diagram of the project. Notice that I<sup>2</sup>C pins of the port expander are connected to pins PB8 and PB9 of the Nucleo-L476RG board and are pulled-up using 10-kohm resistors as required by the I<sup>2</sup>C specifications. The LED is connected to port pin GPA0 of the MCP23017 (pin 21). The address select bits of the MCP23017 are all connected to ground.

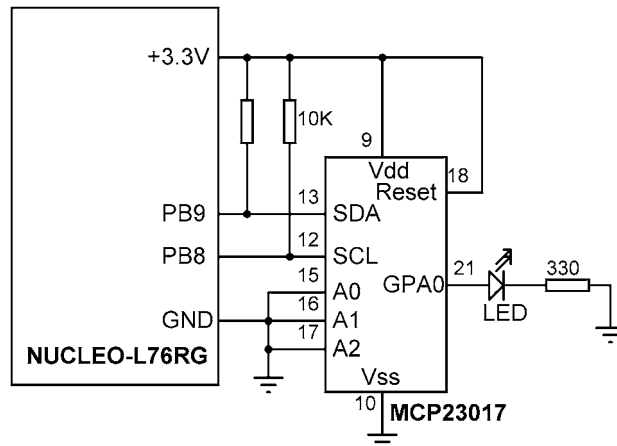


Figure 12.5: Circuit diagram of the project.

Figure 12.6 shows the circuit built on a breadboard. Connections to the Nucleo-L476RG board were made using jumper wires.

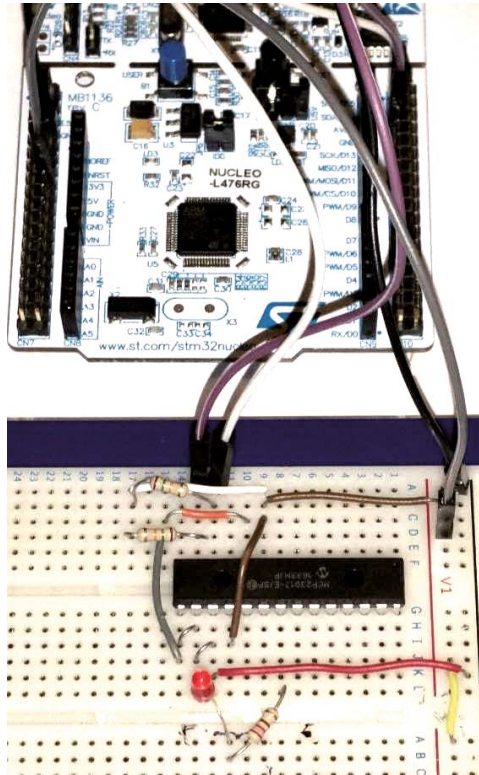


Figure 12.6: Circuit built on a breadboard.

More information on the MCP23017 chip can be obtained from the datasheet at:  
<http://docs-europe.electrocomponents.com/webdocs/137e/0900766b8137eed4.pdf>

### Program listing

The steps are as follows.

- Start the STM32CubeIDE program.
- Select the Nucleo-L476RG board as before.
- Create a new workspace.
- Name the program as **I2CLED**.
- Click on pins PB8 and PB9 on the pin diagram in the middle of the screen and select I2C1\_SCL and I2C1\_SDA as shown in Figure 12.7.

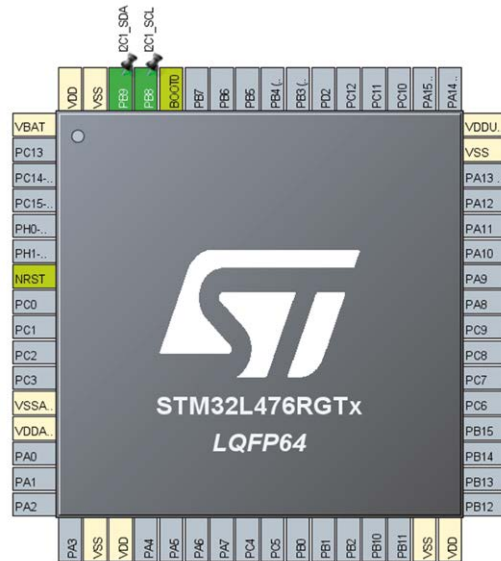


Figure 12.7: Configuring PB8 and PB9 as I<sup>2</sup>C pins.

- Click tab **Connectivity**, and then click on pin **I2C1** at the left hand side and enable **I2C**.
- In **Parameter Settings**, make sure that the **I2C Speed Mode** is set to **Standard Mode** and that the **I2C Speed Frequency** is set to 100 kHz.
- Click **Clock Configuration** and make sure the system clock as well as the clock to I2C1 bus have been configured to 80 MHz.
- Click **File**, followed by **Save** and then click **YES** to generate code.
- Click **Core**, followed by **Src** and double click **main.c** to open main program.

The device and register addresses of the MCP2307 chip are defined at the beginning of the program. You will notice that the following lines of code are added to the program relating to I<sup>2</sup>C:

```
I2C_HandleTypeDef hi2c1;
static void MX_I2C1_Init(void);
MX_I2C1_Init();
```

In addition, the following function is added relating to I<sup>2</sup>C:

```
static void MX_I2C1_Init(void)
{
 hi2c1.Instance = I2C1;
 hi2c1.Init.Timing = 0x10909CEC;
 hi2c1.Init.OwnAddress1 = 0;
 hi2c1.Init.AddressingMode = I2C_ADDRESSINGMODE_7BIT;
 hi2c1.Init.DualAddressMode = I2C_DUALADDRESS_DISABLE;
```



```

hi2c1.Init.OwnAddress2 = 0;
hi2c1.Init.OwnAddress2Masks = I2C_OA2_NOMASK;
hi2c1.Init.GeneralCallMode = I2C_GENERALCALL_DISABLE;
hi2c1.Init.NoStretchMode = I2C_NOSTRETCH_DISABLE;
if (HAL_I2C_Init(&hi2c1) != HAL_OK)
{
 Error_Handler();
}

if (HAL_I2CEx_ConfigAnalogFilter(&hi2c1, I2C_ANALOGFILTER_ENABLE) != HAL_OK)
{
 Error_Handler();
}

if (HAL_I2CEx_ConfigDigitalFilter(&hi2c1, 0) != HAL_OK)
{
 Error_Handler();
}
}

```

A function called **SEND** has been created to send configuration data or application data to the MCP23017 port expander chip. The contents of this file is given below. Notice that I2C function **HAL\_I2C\_Master\_Transmit** sends data over the I<sup>2</sup>C bus. Here, **I2C\_ADDRESS** is the I2C address of the chip (0x40), **buff** is the buffer where the data to be sent is stored, the number of bytes to be sent is 2, and the timeout is set to 1000:

```

void SEND(char port, char data)
{
 uint8_t buff[2];
 buff[0] = port;
 buff[1] = data;
 HAL_I2C_Master_Transmit(&hi2c1, I2C_ADDRESS, buff, 2, 1000);
}

```

Enter the following statements to start the I<sup>2</sup>C and also to configure the MCP23017 chip GPIOA port pin 0 as output.

```

HAL_I2C_Init(&hi2c1); // Start I2C
SEND(IODIRA, 0xFE); // Configure GPA0 as output

```

Inside the main program loop port GPIOA of the MCP23017 chip is toggled every second so that the LED flashes every second.

```

while (1)
{
 SEND(MCP_GPIOA, 0); // LED OFF
 HAL_Delay(1000); // 1 second delay
 SEND(MCP_GPIOA, 1); // LED ON
 HAL_Delay(1000); // 1 second delay
}

```

Figure 12.8 shows the program (program: I2CLED) listing (the clock and error routines are not shown). Compile the program in Release mode and drag and drop the binary file **I2CLED.bin** to device NUCLEO\_L476RG.

```

/* USER CODE BEGIN Header */
/**

 * @file : main.c
 * @brief : Main program body

 * @attention
 *
 * <h2><center>© Copyright (c) 2020 STMicroelectronics.
 * All rights reserved.</center></h2>
 *
 * This software component is licensed by ST under BSD 3-Clause license,
 * the «License»; You may not use this file except in compliance with the
 * License. You may obtain a copy of the License at:
 *
 * opensource.org/licenses/BSD-3-Clause
 *

 */
#include «main.h»

#define IODIRA 0x00
#define MCP_GPIOA 0x12
char I2C_ADDRESS = 0x40;

I2C_HandleTypeDef hi2c1;

void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_I2C1_Init(void);

//
// This function sends configuration or application data to the MCP23017 chip
//
void SEND(char port, char data)

```

```
{
 uint8_t buff[2];
 buff[0] = port;
 buff[1] = data;
 HAL_I2C_Master_Transmit(&hi2c1, I2C_ADDRESS, buff, 2, 1000);
}

int main(void)
{
 HAL_Init();

 SystemClock_Config();

 MX_GPIO_Init();
 MX_I2C1_Init();
 HAL_I2C_Init(&hi2c1); // Start I2C
 SEND(IODIRA, 0xFE); // Configure as outputs

 while (1)
 {
 SEND(MCP_GPIOA, 0); // LED OFF
 HAL_Delay(1000); // 1 second delay
 SEND(MCP_GPIOA, 1); // LED ON
 HAL_Delay(1000); // 1 second delay
 }
}

void SystemClock_Config(void)
{
 RCC_OscInitTypeDef RCC_OscInitStruct = {0};
 RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};
 RCC_PeriphCLKInitTypeDef PeriphClkInit = {0};

 RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
 RCC_OscInitStruct.HSISState = RCC_HSI_ON;
 RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
 RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
 RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
 RCC_OscInitStruct.PLL.PLLM = 2;
 RCC_OscInitStruct.PLL.PLLN = 20;
 RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV7;
 RCC_OscInitStruct.PLL.PLLQ = RCC_PLLQ_DIV2;
 RCC_OscInitStruct.PLL.PLLR = RCC_PLLR_DIV2;
 if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
```

```

{
 Error_Handler();
}

RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYCLK
 |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
RCC_ClkInitStruct.AHBCLKDivider = RCC_SYCLK_DIV1;
RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_4) != HAL_OK)
{
 Error_Handler();
}
PeriphClkInit.PeriphClockSelection = RCC_PERIPHCLK_I2C1;
PeriphClkInit.I2c1ClockSelection = RCC_I2C1CLKSOURCE_PCLK1;
if (HAL_RCCEx_PeriphCLKConfig(&PeriphClkInit) != HAL_OK)
{
 Error_Handler();
}

if (HAL_PWREx_ControlVoltageScaling(PWR_REGULATOR_VOLTAGE_SCALE1) != HAL_OK)
{
 Error_Handler();
}
}

static void MX_I2C1_Init(void)
{
 hi2c1.Instance = I2C1;
 hi2c1.Init.Timing = 0x10909CEC;
 hi2c1.Init.OwnAddress1 = 0;
 hi2c1.Init.AddressingMode = I2C_ADDRESSINGMODE_7BIT;
 hi2c1.Init.DualAddressMode = I2C_DUALADDRESS_DISABLE;
 hi2c1.Init.OwnAddress2 = 0;
 hi2c1.Init.OwnAddress2Masks = I2C_OA2_NOMASK;
 hi2c1.Init.GeneralCallMode = I2C_GENERALCALL_DISABLE;
 hi2c1.Init.NoStretchMode = I2C_NOSTRETCH_DISABLE;
 if (HAL_I2C_Init(&hi2c1) != HAL_OK)
 {
 Error_Handler();
 }

 if (HAL_I2CEx_ConfigAnalogFilter(&hi2c1, I2C_ANALOGFILTER_ENABLE) != HAL_OK)
 {

```

```
 Error_Handler();
}

if (HAL_I2CEx_ConfigDigitalFilter(&hi2c1, 0) != HAL_OK)
{
 Error_Handler();
}
}

static void MX_GPIO_Init(void)
{
 /* GPIO Ports Clock Enable */
 __HAL_RCC_GPIOB_CLK_ENABLE();
}

void Error_Handler(void)
{
}

#ifdef USE_FULL_ASSERT

void assert_failed(uint8_t *file, uint32_t line)
{
}

#endif

/***** (C) COPYRIGHT STMicroelectronics *****/
```

*Figure 12.8: The I2CLED program.*

The following HAL functions are available for I<sup>2</sup>C programming:

**Master to send and receive data in polling mode:**

```
HAL_I2C_Master_Transmit(I2C_HandleTypeDef *hi2c, uint16_t DevAddress,
uint8_t *pData, uint16_t Size, uint32_t Timeout);
HAL_I2C_Master_Receive(I2C_HandleTypeDef *hi2c, uint16_t DevAddress, uint8_t
*pData, uint16_t Size, uint32_t Timeout);
```

**Master to send and receive data in interrupt mode:**

```

HAL_I2C_Master_Transmit_IT(I2C_HandleTypeDef *hi2c, uint16_t DevAddress,
uint8_t *pData, uint16_t Size);
HAL_I2C_Master_Receive_IT(I2C_HandleTypeDef *hi2c, uint16_t DevAddress,
uint8_t *pData, uint16_t Size);

```

**Master to send and receive data in DMA mode:**

```

HAL_I2C_Master_Transmit_DMA(I2C_HandleTypeDef *hi2c, uint16_t DevAddress,
uint8_t *pData, uint16_t Size);
HAL_I2C_Master_Receive_DMA(I2C_HandleTypeDef *hi2c, uint16_t DevAddress,
uint8_t *pData, uint16_t Size);

```

**12.5 Project 2: EEPROM memory****Description**

In this project we will be using the I<sup>2</sup>C bus compatible 24LC256 type EEPROM memory chip and write the BCD bytes '10' in the first two bytes of the memory. The data at the same bytes are then read and converted into an integer number and this number is used as a count to flash the on-board LED.

**The aim**

The aim of this project is to show how an I<sup>2</sup>C based memory can be programmed using the STM32CubeIDE software.

**The 24LC256 memory**

The 24LC256 is a 32 Kbit × 8 (i.e. 256 Kbit) EEPROM memory chip manufactured by Microchip Technology Inc. The chip can operate from 1.7 V to 5.5 V, having standby current of 1 µA and a write current of 3 mA. The chip can operate from 100 kHz to up to 1 MHz. A hardware write protect pin is provided to disable writing to the chip. The 24LC256 is capable of both random and sequential reads up to 256K boundary. The device has page a write capability up to 64 bytes of data. The device has 32768 addresses, ranging from 0x0000 to 0x7FFF. Figure 12.9 shows the pin layout of the chip.

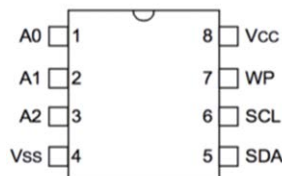


Figure 12.9: Pin assignment of the 24LC256.

A0, A1 and A2 are used to set the LSB bits of the device I<sup>2</sup>C address. As shown below, the upper 4 bits of the device address are fixed at 1010 and the LSB bit is the R/W bit:

|   |   |   |   |    |    |    |     |
|---|---|---|---|----|----|----|-----|
| 1 | 0 | 1 | 0 | A2 | A1 | A0 | R/W |
|---|---|---|---|----|----|----|-----|

For example, if  $A2 = A1 = A0 = 0$  then the I<sup>2</sup>C address is 0xA0.

V<sub>CC</sub> and V<sub>SS</sub> (V<sub>CC</sub>; V<sub>SS</sub>) are the power supply pins.

WP is the write protection pin. If this pin is tied to the ground, writing is enabled. If connected to V<sub>CC</sub> then the write operations have no effects.

### Circuit diagram

The circuit diagram of the project is shown in Figure 12.10. In this project, I<sup>2</sup>C pins PB8 and PB9 of the Nucleo board are used. A0, A1 and A2 are connected to ground so that the device address is 0xA0. Also, the write protect pin WP is tied to ground. The on-board LED at port PA5 is used in the project.

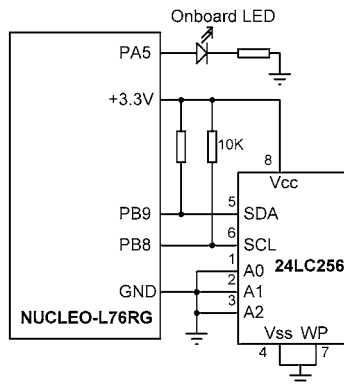


Figure 12.10: Circuit diagram of the project.

### Program listing

The steps are as follows.

- Start the STM32CubeIDE program.
- Select the Nucleo-L476RG board as before.
- Create a new workspace.
- Name the program as **EEPROM**.
- Configure PA5 as digital output. This is where the on-board LED is connected to.
- Click on pins PB8 and PB9 on the pin diagram in the middle of the screen and select I2C1\_SCL and I2C1\_SDA as in the previous project.
- Click tab **Connectivity**, and then click on pin **I2C1** at the left-hand side and enable **I2C**.
- In **Parameter Settings**, make sure that the **I2C Speed Mode** is set to **Standard Mode** and that the **I2C Speed Frequency** is set to 100 kHz.
- Click **Clock Configuration** and make sure the system clock as well as the clock to I2C1 bus have been configured to 80 MHz.
- Click **File**, followed by **Save** and then click **YES** to generate code
- Click **Core**, followed by **Src** and double click **main.c** to open main program

The LED connection and the 24LC256 I2C device address are defined at the beginning of the program.

Before going into details of the memory write and read operations, it is worthwhile to learn how this is done.

### Memory write operation

As an example, assume that we want to write byte 0x25 into memory location 0x0250. Figure 12.11 shows the write steps in detail. First of all, the START bit is sent on the bus, followed by the device address which is assumed to be 0xA0, with the LSB bit set to 0 to indicate that we wish to do a write operation. The memory address 0x0250 is then split into upper and lower bytes as 0x02 and 0x50 and they are sent sequentially with the higher byte sent first over the bus. Then, the data byte 0x25 is sent (this is called **Byte Writing** since only one byte is written to memory). Notice that we can send multiple bytes (this is called **Page Writing** where up to 64 bytes can be written sequentially) in the same transaction (an internal address counter is incremented automatically after a byte is sent). The write operation is terminated with the STOP bit. Notice that ACK bit is sent by the EEPROM between the byte transfers. After a byte write command, the internal address counter will point to the address location following the one that was just written.

Notice that the data sent the EEPROM is stored in a temporary buffer since a whole page consisting of 64 bytes is refreshed after every write operation. It is therefore important to detect when a write operation has been completed successfully. This is done by the master sending a write operation with control byte where R/W set to 0, and making sure that HAL\_OK is returned. If the device is still busy with the write cycle, then no ACK will be returned. If no ACK is returned, the Start bit and control byte must be resent. If the cycle is complete, then the device will return the ACK and the master can then proceed with the next read or write command.

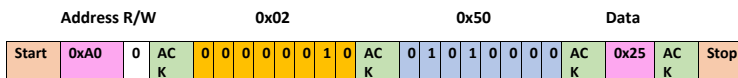


Figure 12.11: Memory Byte Writing operation.

### Memory read operation

Memory read operations are slightly more complex. There are 3 types of reads: current address read, random read, and sequential read. Random read mode is probably the most commonly used mode where the master can access any memory location in a random manner.

As an example, let's assume that we want to read the byte at memory location 0x0250 (where 0x25 was stored in Figure 12.11). Figure 12.12 shows the read steps in detail. To perform random read, the memory address must be sent first. This is done by sending the memory address to the 24LC256 as part of a write operation (R/W bit set to '0'). Once the memory address is sent, the master generates a START condition following the ACK. This terminates the write operation, but not before the internal address counter is set. The master then issues the slave address again, but with the R/W bit set to a 1. The 24LC256 will then issue an ACK and transmit the 8-bit data word. The master will not acknowledge the transfer, though it generates a STOP condition, which causes the EEPROM to discontinue



transmission. After a random read command, the internal address counter will point to the address location following the one that was just read.

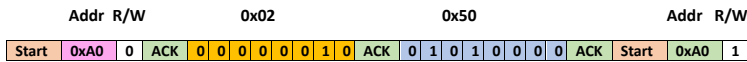


Figure 12.12: Random memory read operation.

Two functions are created in the program to write and read from the memory. Function **WRITE** writes data to the specified address of the memory and it contains the following statements:

```
void WRITE(uint16_t MemLoc, uint8_t *pData, uint16_t len)
{
 uint8_t data[5];
 data[0] = (uint8_t) ((MemLoc & 0xFF00) >> 8);
 data[1] = (uint8_t) (MemLoc & 0xFF);
 memcpy(data+2, pData, len);
 HAL_I2C_Master_Transmit(&hi2c1, I2C_ADDRESS, data, len + 2, HAL_MAX_DELAY);
 while(HAL_I2C_Master_Transmit(&hi2c1, I2C_ADDRESS, 0, 0, HAL_MAX_DELAY) !=
 HAL_OK);
}
```

The function extracts the upper and the lower bytes of the address (**MemLoc**) where we want to write to (address 0x1000) and stores them in **data[0]** and **data[1]** respectively. The data to be written is then appended to data and is sent to the memory using function **HAL\_I2C\_Master\_Transmit**. The last statement inside the function waits until the data has successfully been written to the memory. The data in array **wmsg[0]** (i.e. '1') is written to address 0x1000 and the data in **wmsg[1]** (i.e. '0') is written to address 0x1001. Function **READ** reads bytes from the specified memory locations (starting from address 0x1000) and contains the following statements:

```
void READ(uint16_t MemLoc, uint8_t *pData, uint16_t len)
{
 uint8_t addr[2];
 addr[0] = (uint8_t) ((MemLoc & 0xFF00) >> 8);
 addr[1] = (uint8_t) (MemLoc & 0xFF);
 HAL_I2C_Master_Transmit(&hi2c1, I2C_ADDRESS, addr, 2, HAL_MAX_DELAY);
 HAL_I2C_Master_Receive(&hi2c1, I2C_ADDRESS, pData, len, HAL_MAX_DELAY);
}
```

As with the **WRITE** function, the **READ** function extracts the upper and the lower bytes of the address (**MemLoc**) where we want to read from (address 0x1000) and stores them in **data[0]** and **data[1]** respectively. then, **HAL\_I2C\_Master\_Transmit** is called to send the required memory addresses to the memory. Function **HAL\_I2C\_Master\_Receive** reads two bytes from the specified memory locations.

The **WRITE** and **READ** functions are called in the main program as follows:

```
WRITE(0x1000, (uint8_t*)wmsg, strlen(wmsg)+1); // Write data
READ(0x1000, (uint8_t*)rmsg, strlen(rmsg)+1); // Read data
```

The data is read after 5 seconds. Variable `count` extracts the data from buffer `rmsg` and converts it into an integer. A **for** loop is then formed which toggles the LED by the number of times specified in variable `count`.

Figure 12.13 shows the program listing. Compile the program in Release mode and drag and drop the binary file **EEPROM.bin** to device NUCLEO\_L476RG. The on-board LED should flash 10 times.

```
/* USER CODE BEGIN Header */
/**
 * *****
 * @file : main.c
 * @brief : Main program body
 * *****
 * @attention
 *
 * <h2><center>© Copyright (c) 2020 STMicroelectronics.
 * All rights reserved.</center></h2>
 *
 * This software component is licensed by ST under BSD 3-Clause license,
 * the «License»; You may not use this file except in compliance with the
 * License. You may obtain a copy of the License at:
 *
 * opensource.org/licenses/BSD-3-Clause
 *
 * *****
 */
#include «main.h»
#include «string.h»

#define LED GPIO_PIN_5
char I2C_ADDRESS = 0xA0;

I2C_HandleTypeDef hi2c1;

void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_I2C1_Init(void);

//
// This function reads data from the specified memory locations MemLoc
// The upper and lower memory addresses are stored in data[0] and data[1]
//
```

```
void READ(uint16_t MemLoc, uint8_t *pData, uint16_t len)
{
 uint8_t addr[2];
 addr[0] = (uint8_t) ((MemLoc & 0xFF00) >> 8);
 addr[1] = (uint8_t) (MemLoc & 0xFF);
 HAL_I2C_Master_Transmit(&hi2c1, I2C_ADDRESS, addr, 2, HAL_MAX_DELAY);
 HAL_I2C_Master_Receive(&hi2c1, I2C_ADDRESS, pData, len, HAL_MAX_DELAY);
}

//
// This function writes to the specified memory locations. The upper and lower
// addresses are stored in data[0] and data[1]. The data to be written to memory
// is appended to the data array
//
void WRITE(uint16_t MemLoc, uint8_t *pData, uint16_t len)
{
 uint8_t data[5];

 data[0] = (uint8_t) ((MemLoc & 0xFF00) >> 8);
 data[1] = (uint8_t) (MemLoc & 0xFF);
 memcpy(data+2, pData, len);
 HAL_I2C_Master_Transmit(&hi2c1, I2C_ADDRESS, data, len + 2, HAL_MAX_DELAY);
 while(HAL_I2C_Master_Transmit(&hi2c1, I2C_ADDRESS, 0, 0, HAL_MAX_DELAY) !=
HAL_OK);
}

int main(void)
{
 int count, i;
 char wmsg[] = {'1','0'}; // Data to be written
 char rmsg[10]; // Data read

 HAL_Init();

 SystemClock_Config();

 MX_GPIO_Init();
 MX_I2C1_Init();
 HAL_I2C_Init(&hi2c1); // Start I2C

 WRITE(0x1000, (uint8_t*)wmsg, strlen(wmsg)+1); // Write data

 HAL_Delay(5000); // Wait 5 secs

 READ(0x1000, (uint8_t*)rmsg, strlen(rmsg)+1); // Read data
```

```

count = 10*(rmsg[0] - '0') + rmsg[1]-'0'; // Calculate count

for(i = 0; i < count; i++)
{
 HAL_GPIO_TogglePin(GPIOA, LED); // Toggle LED
 HAL_Delay(1000);
}

while(1)
{
}
}

void SystemClock_Config(void)
{
 RCC_OscInitTypeDef RCC_OscInitStruct = {0};
 RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};
 RCC_PeriphCLKInitTypeDef PeriphClkInit = {0};

 RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
 RCC_OscInitStruct.HSISState = RCC_HSI_ON;
 RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
 RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
 RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
 RCC_OscInitStruct.PLL.PLLM = 2;
 RCC_OscInitStruct.PLL.PLLN = 20;
 RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV7;
 RCC_OscInitStruct.PLL.PLLQ = RCC_PLLQ_DIV2;
 RCC_OscInitStruct.PLL.PLLR = RCC_PLLR_DIV2;
 if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
 {
 Error_Handler();
 }

 RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
 |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
 RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
 RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
 RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
 RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

 if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_4) != HAL_OK)
 {
 Error_Handler();
 }
 PeriphClkInit.PeriphClockSelection = RCC_PERIPHCLK_I2C1;

```

```
PeriphClkInit.I2c1ClockSelection = RCC_I2C1CLKSOURCE_PCLK1;
if (HAL_RCCEx_PeriphCLKConfig(&PeriphClkInit) != HAL_OK)
{
 Error_Handler();
}

if (HAL_PWREx_ControlVoltageScaling(PWR_REGULATOR_VOLTAGE_SCALE1) != HAL_OK)
{
 Error_Handler();
}
}

static void MX_I2C1_Init(void)
{
 hi2c1.Instance = I2C1;
 hi2c1.Init.Timing = 0x10909CEC;
 hi2c1.Init.OwnAddress1 = 0;
 hi2c1.Init.AddressingMode = I2C_ADDRESSINGMODE_7BIT;
 hi2c1.Init.DualAddressMode = I2C_DUALADDRESS_DISABLE;
 hi2c1.Init.OwnAddress2 = 0;
 hi2c1.Init.OwnAddress2Masks = I2C_OA2_NOMASK;
 hi2c1.Init.GeneralCallMode = I2C_GENERALCALL_DISABLE;
 hi2c1.Init.NoStretchMode = I2C_NOSTRETCH_DISABLE;
 if (HAL_I2C_Init(&hi2c1) != HAL_OK)
 {
 Error_Handler();
 }

 if (HAL_I2CEx_ConfigAnalogFilter(&hi2c1, I2C_ANALOGFILTER_ENABLE) != HAL_OK)
 {
 Error_Handler();
 }

 if (HAL_I2CEx_ConfigDigitalFilter(&hi2c1, 0) != HAL_OK)
 {
 Error_Handler();
 }
}

static void MX_GPIO_Init(void)
{
 GPIO_InitTypeDef GPIO_InitStruct = {0};

 /* GPIO Ports Clock Enable */
 __HAL_RCC_GPIOA_CLK_ENABLE();
 __HAL_RCC_GPIOB_CLK_ENABLE();
```

```

/*Configure GPIO pin Output Level */
HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_RESET);

/*Configure GPIO pin : PA5 */
GPIO_InitStruct.Pin = GPIO_PIN_5;
GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
GPIO_InitStruct.Pull = GPIO_NOPULL;
GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
}

void Error_Handler(void)
{

}

#ifdef USE_FULL_ASSERT

void assert_failed(uint8_t *file, uint32_t line)
{

}

#endif

/***** (C) COPYRIGHT STMicroelectronics *****/

```

*Figure 12.13: The EEPROM program.*

### Modified program

The I<sup>2</sup>C memory type operations are very common. As a result of this, special HAL functions have been developed by STMicroelectronics that can be used to simplify the memory operations. These functions are:

#### To write to memory:

```
HAL_I2C_Mem_Write(handle, DeviceAddress, uint16_t MemLoc, uint16_t MemAddSize,
uint8_t *pData, uint16_t Size, uint32_t Timeout);
```

#### To read from memory:

```
HAL_I2C_Mem_Read(handle, DeviceAddress, uint16_t MemLoc, uint16_t MemAddSize,
uint8_t *pData, uint16_t Size, uint32_t Timeout);
```

Additionally, a function is provided to check whether or not the memory write operation is complete:

```
HAL_I2C_IsDeviceReady(handle, DeviceAddress, uint32_t Trials, uint32_t Timeout);
```

Where Trials is the number of trials to make before returning an error condition (you should make the Timeout to be HAL\_MAX\_DELAY so that you can chose any value for Trials)

Using the new functions, the program in Figure 12.13 can be simplified as shown in Figure 12.14, where the clock, error, and I/O initialization functions are not shows (all other parts of the two programs are identical).

```
#include «main.h»
#include «string.h»

#define LED GPIO_PIN_5
char I2C_ADDRESS = 0xA0;

I2C_HandleTypeDef hi2c1;

void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_I2C1_Init(void);

int main(void)
{
 int count, i;
 char wmsg[] = {'1','0'}; // Data to be written
 char rmsg[10]; // Data read

 HAL_Init();

 SystemClock_Config();

 MX_GPIO_Init();
 MX_I2C1_Init();
 HAL_I2C_Init(&hi2c1); // Start I2C

 //
 // Write data to memory
 //
 HAL_I2C_Mem_Write(&hi2c1,I2C_ADDRESS, 0x1000, I2C_MEMADD_SIZE_16BIT, (uint8_t*)
wmsg,
 strlen(wmsg)+1, HAL_MAX_DELAY);

 //
 // Wait for write complete (Trials is set to 1)
 //
 while(HAL_I2C_IsDeviceReady(&hi2c1, I2C_ADDRESS, 1, HAL_MAX_DELAY) != HAL_OK);

 HAL_Delay(5000); // Wait 5 secs
```

```
//
// Read from memory
//
HAL_I2C_Mem_Read(&hi2c1, I2C_ADDRESS, 0x1000, I2C_MEMADD_SIZE_16BIT, (uint8_t*)
rmsg,
strlen(wmsg)+1, HAL_MAX_DELAY);

count = 10*(rmsg[0] - '0') + rmsg[1]-'0'; // Calculate count

for(i = 0; i < count; i++)
{
 HAL_GPIO_TogglePin(GPIOA, LED); // Toggle LED
 HAL_Delay(1000);
}

while(1)
{
}
}
```

*Figure 12.14: Modified, simpler program.*

## 12.6 Project 3: TMP102 Temperature Sensor Chip Reading

### Description

In this project the I<sup>2</sup>C compatible TMP102 temperature sensor chip is used. The ambient temperature is read every second and is sent to the PC through the USART where it is displayed on the terminal emulation screen.

### The aim

The aim of this project is to show how the temperature sensor chip TMP102 can be used in a program together with the USART.

### The TMP102

The TMP102 is an I<sup>2</sup>C compatible temperature sensor chip having the following basic features:

- Supply voltage: 1.4 V to 3.6 V
- Supply current: 10  $\mu$ A
- Accuracy: 2  $^{\circ}$ C
- Resolution: 12 bits (0.0625  $^{\circ}$ C)
- Accuracy:  $\pm 0.5$   $^{\circ}$ C

The TMP102 is a 6-pin chip as shown in Figure 12.15. the pin descriptions are:



| Pin | Name  | Description                                                            |
|-----|-------|------------------------------------------------------------------------|
| 1   | SCL   | I <sup>2</sup> C line                                                  |
| 2   | GND   | Power supply ground                                                    |
| 3   | ALERT | Over temperature alert. Open-drain output. requires a pull-up resistor |
| 4   | ADD0  | Address select                                                         |
| 5   | V+    | Power supply                                                           |
| 6   | SDA   | I <sup>2</sup> C line                                                  |

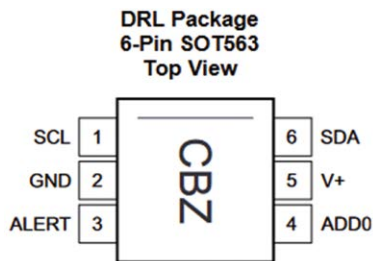


Figure 12.15: TMP102 pin assignment.

The default operating mode of TMP102 is continuous conversion where the internal ADC converts the temperature into digital with the default rate of 4Hz, with a conversion time of 26ms. The temperature register is a 16-bit register where bits 0, 1, 2, and 3 are set to 0, and bits 4 to 15 store the 12-bit temperature data.

TMP102 is available as a module (breakout module) as shown in Figure 12.16. the default device address is 0x48. The temperature register address is 0x00 and this should be sent after sending the device address. This is then followed with a read command where two bytes are read from the TMP102. These bytes contain the temperature data.

The temperature read sequence is as follows.

- Master sends the device address 0x48 with the R/W set to 0.
- Device responds with ACK.
- Master sends the temperature register address 0x00.
- Device responds with ACK.
- Master re-sends device address 0x48 with the R/W bit set to 1.
- Master reads upper byte of temperature data.
- Device sends ACK.
- Master reads lower byte of temperature data.
- Device sends ACK.
- Master sends stop condition on the bus.

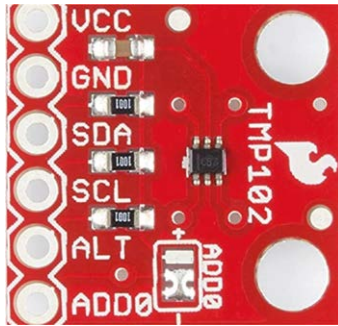


Figure 12.16 TMP102 as a module

### Block diagram

Figure 12.17 shows the block diagram of the project.

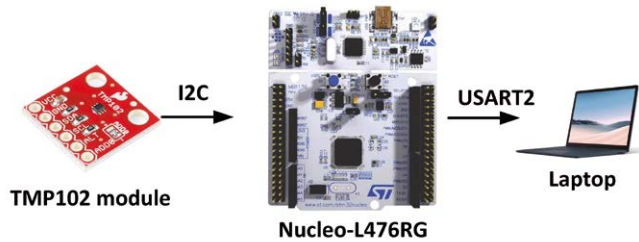


Figure 12.17: Block diagram of the project.

### Circuit diagram

The circuit diagram of the project is shown in Figure 12.18. On-chip pull-up resistors are available on the TMP102 I<sup>2</sup>C bus lines.

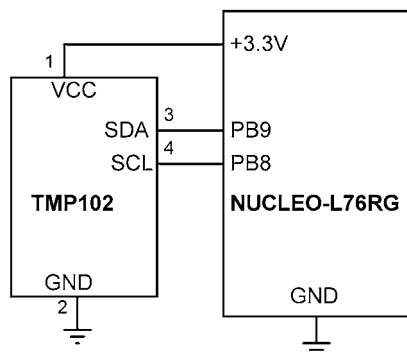


Figure 12.18: Circuit diagram of the project.

### Program listing

In this program PB8 and PB9 I<sup>2</sup>C lines of the Nucleo board are used as in the previous program. Serial data is sent to the PC over the USART2 line so that the temperature can be displayed on the terminal emulator screen.

The steps are given here.

- Start the STM32CubeIDE program.
- Select the Nucleo-L476RG board as before.
- Create a new workspace.
- Name the program as **TMP102**.
- Click on pins PB8 and PB9 on the pin diagram in the middle of the screen and select I2C1\_SCL and I2C1\_SDA as in the previous project.
- Click tab **Connectivity**, and then click on pin **I2C1** at the left-hand side and enable **I2C**.
- In **Parameter Settings**, make sure that the **I2C Speed Mode** is set to **Standard Mode** and that the **I2C Speed Frequency** is set to 100 kHz.
- Click **Clock Configuration** and make sure the system clock as well as the clock to I2C1 bus have been configured to 80 MHz.
- Click **Connectivity**, then click **USART2** and set **Mode** to **Asynchronous**
- In **Parameter Settings**, set the **Baud Rate** to 9600, **Word Length** to 8 Bits, **Parity** to None, and **Stop Bits** to 1.
- Click **File**, followed by **Save** and then click **YES** to generate code.
- Click **Core**, followed by **Src** and double click **main.c** to open main program.

At the beginning of the program the I2 address of TMP102 and the temperature register addresses are defined:

```
uint8_t I2C_ADDRESS = 0x48 << 1;
uint8_t Temp_Reg = 0x00;
```

The following two user functions are used in the program:

```
void UART_SEND_TXT(UART_HandleTypeDef *huart, char buffer[], int m)void
UART_SEND_NL(UART_HandleTypeDef *huart)
```

The program runs inside a **while** loop. Here, the temperature is read from TMP102 and stored in variable **comb**. If the temperature is negative, then it is in 2's complement form and its complement is taken and 1 is added to find the true negative value. By multiplying **comb** with the LSB we find the temperature in degrees Centigrade. The temperature is then converted into a string and stored in character array called **buf**. Finally, the temperature reading is sent to the PC through the UART and is displayed on the terminal emulator screen as shown in Figure 12.19. The above process is repeated after one second delay. Table 12.1 shows the data output format of the temperature. Let's look at two examples:

**Example 1:** Measured value = 0011 00100000 = 0x320 = 800 decimal  
This is positive temperature, so the temperature is  $800 \times 0.0625 = +50\text{ }^{\circ}\text{C}$

**Example 2:** Measured value = 1110 01110000 = 0xE70  
This is negative temperature; the complement being 0001 10001111, adding 1 gives 0001 10010000 = 400 decimal. The temperature is  $400 \times 0.0625 = 25$  or  $-25\text{ }^{\circ}\text{C}$ .

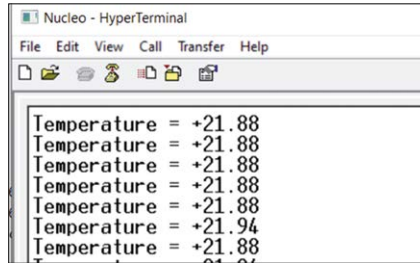


Figure 12.19: Example output from the program.

Table 12.1 shows the data output for some temperature readings.

| TEMPERATURE | DIGITAL OUTPUT (BINARY) | DIGITAL OUTPUT (HEX) |
|-------------|-------------------------|----------------------|
| 128         | 011111111111            | 7FF                  |
| 100         | 011001000000            | 640                  |
| 50          | 001100100000            | 320                  |
| 0.25        | 000000000100            | 004                  |
| -0.25       | 111111111100            | FFC                  |
| -25         | 111001110000            | E70                  |
| -55         | 110010010000            | C90                  |

Table 12.1: Examples of data output resulting from temperature measurements.

Figure 12.20 shows the program listing (Program name: TMP102).

```

/* USER CODE BEGIN Header */
/**
 * *****
 * @file : main.c
 * @brief : Main program body
 * *****
 * @attention
 *
 * <h2><center>© Copyright (c) 2020 STMicroelectronics.
 * All rights reserved.</center></h2>
 *
 * This software component is licensed by ST under BSD 3-Clause license,
 * the «License»; You may not use this file except in compliance with the
 * License. You may obtain a copy of the License at:
 *
 * opensource.org/licenses/BSD-3-Clause
 *
 * *****
 */
#include «main.h»
#include «string.h»

```

```
#include «stdio.h»

uint8_t I2C_ADDRESS = 0x48 << 1;
uint8_t Temp_Reg = 0x00;

I2C_HandleTypeDef hi2c1;

UART_HandleTypeDef huart2;

void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_I2C1_Init(void);
static void MX_USART2_UART_Init(void);

//
// Send new-line
//
void UART_SEND_NL(UART_HandleTypeDef *huart)
{
 HAL_UART_Transmit(huart, (uint8_t*)»\n\r«, 2, HAL_MAX_DELAY);
}

//
// Send text
//
void UART_SEND_TXT(UART_HandleTypeDef *huart, char buffer[], int m)
{
 HAL_UART_Transmit(huart, (uint8_t*) buffer, strlen(buffer), HAL_MAX_DELAY);
 if(m == 1) HAL_UART_Transmit(huart, (uint8_t*)»\n\r«, 2, HAL_MAX_DELAY);
}

//
// Start of main program
//
int main(void)
{
 uint8_t buf[7];
 int16_t comb;
 float temperature, LSB = 0.0625;

 HAL_Init();

 SystemClock_Config();

 MX_GPIO_Init();
 MX_I2C1_Init();
 // Init GPIO
 // Init I2C
```

```

MX_USART2_UART_Init(); // Init UART2
HAL_I2C_Init(&hi2c1); // Start I2C

while (1)
{
 buf[0] = Temp_Reg;
 HAL_I2C_Master_Transmit(&hi2c1, I2C_ADDRESS, buf, 1, HAL_MAX_DELAY);
 HAL_I2C_Master_Receive(&hi2c1, I2C_ADDRESS, buf, 2, HAL_MAX_DELAY);
// Read 2 bytes
 comb = ((int16_t)buf[0] << 4) | (buf[1] >> 4);
// Combine the bytes
 if (comb > 0x7FFF)
// If negative
 {
 comb = (~comb) & 0xFFFF;
 comb = comb + 1;
 temperature = -comb * LSB;
 }
 else
 temperature = comb * LSB;

 sprintf((char*)buf, «%+5.2f», temperature);
 UART_SEND_TXT(&huart2, «Temperature = », 0);
 HAL_UART_Transmit(&huart2, buf, strlen((char*)buf), HAL_MAX_DELAY);
// Send to PC
 UART_SEND_NL(&huart2);
 HAL_Delay(1000);
// Wait 1 sec
}
}

void SystemClock_Config(void)
{
 RCC_OscInitTypeDef RCC_OscInitStruct = {0};
 RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};
 RCC_PeriphCLKInitTypeDef PeriphClkInit = {0};

 RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
 RCC_OscInitStruct.HSISState = RCC_HSI_ON;
 RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
 RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
 RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
 RCC_OscInitStruct.PLL.PLLM = 2;
 RCC_OscInitStruct.PLL.PLLN = 20;
 RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV7;
 RCC_OscInitStruct.PLL.PLLQ = RCC_PLLQ_DIV2;

```

```
RCC_OscInitStruct.PLL.PLLR = RCC_PLLR_DIV2;
if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
{
 Error_Handler();
}

RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
 |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_4) != HAL_OK)
{
 Error_Handler();
}
PeriphClkInit.PeriphClockSelection = RCC_PERIPHCLK_USART2|RCC_PERIPHCLK_I2C1;
PeriphClkInit.Usart2ClockSelection = RCC_USART2CLKSOURCE_PCLK1;
PeriphClkInit.I2c1ClockSelection = RCC_I2C1CLKSOURCE_PCLK1;
if (HAL_RCCEx_PeriphCLKConfig(&PeriphClkInit) != HAL_OK)
{
 Error_Handler();
}

if (HAL_PWREx_ControlVoltageScaling(PWR_REGULATOR_VOLTAGE_SCALE1) != HAL_OK)
{
 Error_Handler();
}
}

static void MX_I2C1_Init(void)
{
 hi2c1.Instance = I2C1;
 hi2c1.Init.Timing = 0x10909CEC;
 hi2c1.Init.OwnAddress1 = 0;
 hi2c1.Init.AddressingMode = I2C_ADDRESSINGMODE_7BIT;
 hi2c1.Init.DualAddressMode = I2C_DUALADDRESS_DISABLE;
 hi2c1.Init.OwnAddress2 = 0;
 hi2c1.Init.OwnAddress2Masks = I2C_OA2_NOMASK;
 hi2c1.Init.GeneralCallMode = I2C_GENERALCALL_DISABLE;
 hi2c1.Init.NoStretchMode = I2C_NOSTRETCH_DISABLE;
 if (HAL_I2C_Init(&hi2c1) != HAL_OK)
 {
 Error_Handler();
 }
}
```

```

 if (HAL_I2CEx_ConfigAnalogFilter(&hi2c1, I2C_ANALOGFILTER_ENABLE) != HAL_OK)
 {
 Error_Handler();
 }

 if (HAL_I2CEx_ConfigDigitalFilter(&hi2c1, 0) != HAL_OK)
 {
 Error_Handler();
 }
}

static void MX_USART2_UART_Init(void)
{
 huart2.Instance = USART2;
 huart2.Init.BaudRate = 9600;
 huart2.Init.WordLength = UART_WORDLENGTH_8B;
 huart2.Init.StopBits = UART_STOPBITS_1;
 huart2.Init.Parity = UART_PARITY_NONE;
 huart2.Init.Mode = UART_MODE_TX_RX;
 huart2.Init.HwFlowCtl = UART_HWCONTROL_NONE;
 huart2.Init.OverSampling = UART_OVERSAMPLING_16;
 huart2.Init.OneBitSampling = UART_ONE_BIT_SAMPLE_DISABLE;
 huart2.AdvancedInit.AdvFeatureInit = UART_ADVFEATURE_NO_INIT;
 if (HAL_UART_Init(&huart2) != HAL_OK)
 {
 Error_Handler();
 }
}

static void MX_GPIO_Init(void)
{
 /* GPIO Ports Clock Enable */
 __HAL_RCC_GPIOA_CLK_ENABLE();
 __HAL_RCC_GPIOB_CLK_ENABLE();
}

void Error_Handler(void)
{
}

#ifdef USE_FULL_ASSERT

void assert_failed(uint8_t *file, uint32_t line)
{

```



```
}
#endif

/***** (C) COPYRIGHT STMicroelectronics *****/
```

*Figure 12.20: The TMP102 program.*

## 12.7 Summary

In this Chapter we have learned how to use the I<sup>2</sup>C bus by developing several projects. In the next Chapter we will be focusing on the SPI bus, which is another important bus to consider for interfacing sensors and devices to a microcontroller.

## CHAPTER 13 • SPI Bus Projects

### 13.1 Overview

In last Chapter we have learned how to interface the I<sup>2</sup>C devices to our development board, as well as how to program the STM32CubeIDE software for the I<sup>2</sup>C bus.

In this Chapter we shall be developing projects using the SPI bus (Serial Peripheral Interface) with the Nucleo-L476RG development board. The SPI bus is commonly one of the commonly used protocols to connect sensors and many other devices to microcontrollers. The SPI bus is a master-slave type bus protocol. In this protocol, one device (the microcontroller) is designated as the master, and one or more other devices (usually sensors) are designated as slaves. In a minimum bus configuration, there is one master and only one slave. The master establishes communication with the slaves and controls all the activity on the bus.

Figure 13.1 shows an SPI bus example with one master and 3 slaves. The SPI bus uses three signals: clock (SCK), data in (SDI), and data out (SDO). SDO of the master is connected to the SDIs of the slaves, and SDOs of the slaves are connected to the SDI of the master. The master generates the SCK signals to enable data to be transferred on the bus. In every clock pulse one bit of data is moved from master to slave, or from slave to master. The communication is only between a master and a slave, and the slaves cannot communicate with each other. It is important to note that only one slave can be active at any time since there is no mechanism to identify the slaves. Thus, slave devices have enable lines (e.g. CS) which are normally controlled by the master. A typical communication between a master and several slaves is as follows:

- master enables slave 1;
- master sends SCK signals to read or write data to slave 1;
- master disables slave 1 and enables slave 2;
- master sends SCK signals to read or write data to slave 2;
- the above process continues as required.

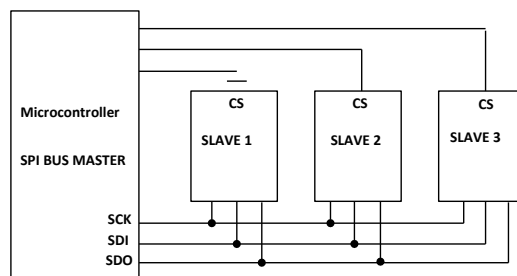


Figure 13.1: SPI bus with one master and three slaves.

The SPI signal names are also called MISO (Master in, Slave out), and MOSI (Master out, Slave in). Clock signal SCK is also called SCLK and the CS is also called SSEL. In the SPI projects in this Chapter the Nucleo-L476RG development board will be the master and one or more slaves will be connected to the bus. Transactions over the SPI bus are started by enabling the SCK line. The master then asserts the SSEL line LOW so that data transmission can begin. The data transmission involves two registers, one in the master and one in

the slave device. Data is shifted out from the master into the slave with the MSB bit first. If more data is to be transferred, then the process is repeated. Data exchange is complete when the master stops sending clock pulses and deselects the slave device.

Both the master and the slave must agree on the clock polarity and phase on the line, which are known as the SPI bus modes. These two settings are named as Clock Polarity (CPOL) and Clock Phase (CPHA) respectively. CPOL and CPHA can have the following values:

| <b>CPOL</b> | <b>Clock active state</b> |
|-------------|---------------------------|
| 0           | Clock active HIGH         |
| 1           | Clock active LOW          |

| <b>CPHA</b> | <b>Clock phase</b>           |
|-------------|------------------------------|
| 0           | Clock out of phase with data |
| 1           | Clock in phase with data     |

The four SPI modes are:

| <b>Mode</b> | <b>CPOL</b> | <b>CPHA</b> |
|-------------|-------------|-------------|
| 0           | 0           | 0           |
| 1           | 0           | 1           |
| 2           | 1           | 0           |
| 3           | 1           | 1           |

When CPOL = 0, the active state of the clock is 1, and its idle state is 0. For CPHA = 0, data is captured on the rising clock, and data is shifted out on the falling clock. For CPHA = 1, data is captured on the falling edge of the clock and gets shifted out on the rising edge of the clock.

When CPOL = 1, the active state of the clock is 0, and its idle state is 1. For CPHA = 0, data is captured on the falling edge of the clock and is output on the rising edge. For CPHA = 1, data is captured on the rising edge of the clock and gets shifted out on the falling edge.

### 13.2 Nucleo-L476RG SPI pins

When creating an SPI bus variable, we must specify the GPIO pins for the MOSI, MISO, and SCLK.

There are three SPI modules on the Nucleo-L476RG development board. The following are the GPIO pins for these modules:

| <b>SPI Module</b> | <b>Signal</b> | <b>GPIO Pin</b> |
|-------------------|---------------|-----------------|
| SPI1              | SSEL          | PA15, PA4       |
| SPI1              | SCLK          | PA5, PB3        |
| SPI1              | MISO          | PA6, PB4        |
| SPI1              | MOSI          | PA7, PB5        |
| SPI2              | MISO          | PC2, PB14       |
| SPI2              | MOSI          | PC3, PB15       |
| SPI2              | SSEL          | PB9, PB12       |

|      |      |            |
|------|------|------------|
| SPI2 | SCLK | PB10, PB13 |
| SPI3 | MOSI | PC12       |
| SPI3 | SCLK | PC10       |
| SPI3 | MISO | PC11       |
| SPI3 | SSEL | PA4        |

### 13.3 Project 1: Port Expander

#### Description

A simple project is given in this section to show how the SPI functions can be used in a program. This project is very similar to the port expander Project 1 in Chapter 12. In that project the I<sup>2</sup>C compatible chip MCP23017 was used. In this project the SPI bus compatible port expander chip MCP23S17 is used to give additional 16 I/O ports to the Nucleo-L476RG board. The operation of the MCP23S17 is same as the MCP23017, except that the MCP23S17 uses the SPI bus. In this project an LED is connected to MCP23S17 port pin GPA0 and the LED is flashed ON and OFF every second. A 330-ohm current limiting resistor is used in series with the LED.

#### The aim

The aim of this project is to show how the SPI bus can be used in Nucleo-L476RG based projects.

#### Block diagram

The block diagram of the project is same as in Figure 12.2, but MCP23017 chip is replaced with MCP23S17.

#### The MCP23S17

The MCP23S17 is a 28-pin chip with the following features. The pin configuration is shown in Figure 13.2, which is same as the pin configuration of MCP23017, but SPI pins are used instead of I<sup>2</sup>C pins:

- 16 bi-directional I/O ports;
- up to 1.7 MHz operation on I<sup>2</sup>C bus;
- interrupt capability;
- external reset input;
- low standby current;
- +1.8 to +5.5 V operation;
- 3 address pins so that up to 8 devices can be used on the SPI bus;
- 28-pin DIL package.

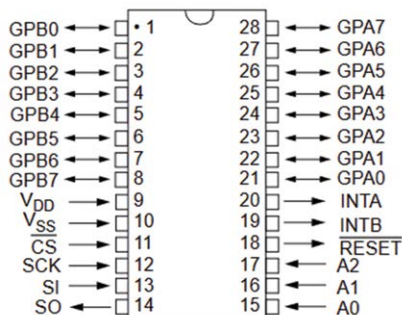


Figure 13.2: Pin configuration of the MCP23S17.

The pin descriptions are given in Table 13.1.

| Pin       | Description              |
|-----------|--------------------------|
| GPA0-GPA7 | Port A pins              |
| GPB0-GPB7 | Port B pins              |
| VDD       | Power supply             |
| VSS       | Ground                   |
| SI        | SPI MOSI data pin        |
| SCK       | SPI clock pin            |
| SO        | SPI MISO data pin        |
| CS        | SPI SSEL chip enable pin |
| A0-A2     | I2C address pins         |
| RESET     | Reset pin                |
| INTA      | Interrupt pin            |
| INTB      | Interrupt pin            |

Table 13.1: MCP23S17 pin descriptions.

The MCP23S17 is a slave SPI device. The slave address contains four upper fixed bits (0100) and three user-defined hardware address bits (pins A2, A1 and A0) with the read/write bit filling out the control byte. These address bits are enabled/disabled by control register IOCON.HAEN. By default, the user address bits are disabled at power-up (i.e. IOCON.HAEN = 0) and A2 = A1 = A0 = 0. Figure 13.3 and Figure 13.4 show the addressing format. The address pins should be externally biased even if disabled.

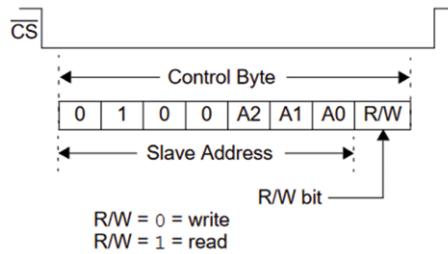


Figure 13.3: MCP23S17 control byte format.

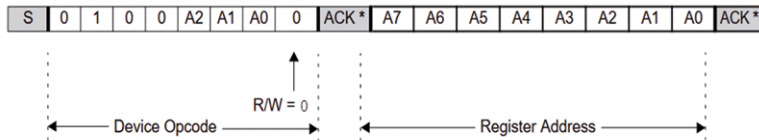


Figure 13.4: MCP23S17 addressing registers.

Like the MCP23017, the MCP23S17 chip has 8 internal registers that can be configured for its operation. The device can either be operated in 16-bit mode or in two 8-bit mode by configuring bit IOCON.BANK. On power-up this bit is cleared which chooses the two 8-bit mode by default.

The I/O direction of the port pins are controlled with registers IODIRA (at address 0x00) and IODIRB (at address 0x01). Clearing a bit to 0 in these registers makes the corresponding port pin(s) as output(s). Similarly, setting a bit to 1 in these registers make the corresponding port pin(s) input(s). GPIOA and GPIOB register addresses are 0x12 and 0x13 respectively. This is shown in Figure 13.5.

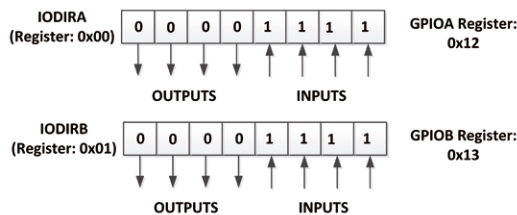


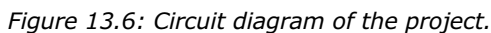
Figure 13.5: Configuring the I/O ports.

Further information on the MCP23S17 chip can be obtained from the Microchip Inc data sheet at the following website:

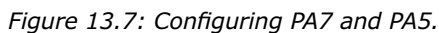
<http://ww1.microchip.com/downloads/en/DeviceDoc/20001952C.pdf>

### Circuit diagram

Figure 13.6 shows the circuit diagram of the project. SPI1 pins PA5 and PA7 are used for the SPI interface. Pin PC0 is used as the MCP23S17 chip enable pin.



- The steps are as follows.
- Start STM32CubeIDE.
- Create a new workspace.
- Give the name **SPILED** to the program.
- Configure **PC0** as digital output.
- Configure **PA7** as **SPI1\_MOSI**, **PA5** as **SPI1\_SCK** (Figure 13.7).



- Click **Connectivity** and then click **SPI1**. Set the **Mode** to **Transmit Only Master**, and **Hardware NSS Signal** to **Disable**.
- In **Parameter Settings**, set the **Data Size** to 8 bits, and **First Bit** to **MSB first** (Figure 13.8).

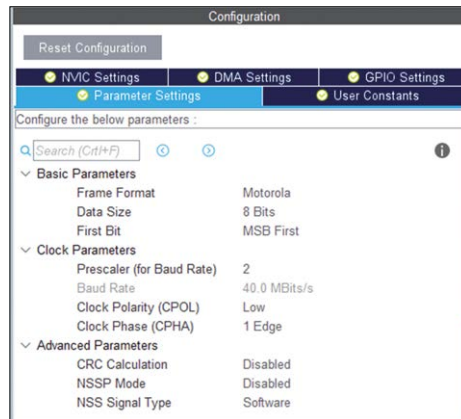


Figure 13.8: Parameter settings.

- Configure the MCU clock for 80 MHz.
- Click **File**, followed by **Save** and click **YES** to generate code.
- Click **Core**, followed by **Src**, and double click **main.c** to open the main program.

Enter the SPI device address and register IODIRA address at the beginning of the program. Also, define the LED connection:

```
#define IODIRA 0x00
#define MCP_GPIOA 0x12
char SPI_ADDRESS = 0x40;
#define CS GPIO_PIN_0
Enable the SPI interface:
HAL_SPI_Init(&hspi1);
```

A function called **SEND** has been created to send data to the slave device over the SPI bus. This function has two parameters: the register address and the data to be loaded into the specified register. Data is sent over the SPI bus using the HAL function **HAL\_SPI\_Transmit**. Notice that the sequence of sending data is as follows: set CS LOW, send device address, send register address, send data, set CS HIGH:

```
void SEND(char RegAddr, char data)
{
 uint8_t buff[3];
 HAL_GPIO_WritePin(GPIOC, CS, GPIO_PIN_RESET) // Enable CS
 buff[0] = SPI_ADDRESS; // SPI Address
 buff[1] = RegAddr; // Register address
 buff[2] = data; // Data
 HAL_SPI_Transmit(&hspi1, buff, 3, 1000); // Send
 HAL_GPIO_WritePin(GPIOC, CS, GPIO_PIN_SET); // Disable CS
}
```



The MCP23S17 is configured using the following function call so that port pin GPIOA is output:

```
SEND(IODIRA, 0xFE); // Set GPIOA as output
```

The program loop toggles port GPIOA of the MCP23S17 every second so that the LED flashes:

```
while (1)
{
 SEND(MCP_GPIOA, 0); // LED OFF
 HAL_Delay(1000); // 1 second delay

 SEND(MCP_GPIOA, 1); // LED ON
 HAL_Delay(1000); // 1 second delay
}
```

Figure 13.9 shows the program listing (Program: SPILED).

```
/* USER CODE BEGIN Header */
/**
 * *****
 * @file : main.c
 * @brief : Main program body
 * *****
 * @attention
 *
 * <h2><center>© Copyright (c) 2020 STMicroelectronics.
 * All rights reserved.</center></h2>
 *
 * This software component is licensed by ST under BSD 3-Clause license,
 * the «License»; You may not use this file except in compliance with the
 * License. You may obtain a copy of the License at:
 *
 * opensource.org/licenses/BSD-3-Clause
 *
 * *****
 */
#include «main.h»
#define IODIRA 0x00
#define MCP_GPIOA 0x12

char SPI_ADDRESS = 0x40; // SPI address
#define CS GPIO_PIN_0 // CS pin

SPI_HandleTypeDef hspi1;
```

```

void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_SPI1_Init(void);

//
// This function sends configuration or application data to the MCP23S17 chip
// The data is sent as: SPI Device Address, Register address, data
//
void SEND(char RegAddr, char data)
{
 uint8_t buff[3];
 HAL_GPIO_WritePin(GPIOC, CS, GPIO_PIN_RESET); // Enable CS
 buff[0] = SPI_ADDRESS; // SPI Address
 buff[1] = RegAddr; // Register address
 buff[2] = data; // Data
 HAL_SPI_Transmit(&hspi1, buff, 3, 1000); // Send
 HAL_GPIO_WritePin(GPIOC, CS, GPIO_PIN_SET); // Disable CS
}

//
// Start of main program
//
int main(void)
{
 HAL_Init();

 SystemClock_Config();

 MX_GPIO_Init();
 MX_SPI1_Init();
 HAL_SPI_Init(&hspi1); // Start SPI

 SEND(IODIRA, 0xFE); // Set GPIOA as output

 while (1)
 {
 SEND(MCP_GPIOA, 0); // LED OFF
 HAL_Delay(1000); // 1 second delay

 SEND(MCP_GPIOA, 1); // LED ON
 HAL_Delay(1000); // 1 second delay
 }
}

void SystemClock_Config(void)
{

```

```
RCC_OscInitTypeDef RCC_OscInitStruct = {0};
RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
RCC_OscInitStruct.HSIState = RCC_HSI_ON;
RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
RCC_OscInitStruct.PLL.PLLM = 2;
RCC_OscInitStruct.PLL.PLLN = 20;
RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV7;
RCC_OscInitStruct.PLL.PLLQ = RCC_PLLQ_DIV2;
RCC_OscInitStruct.PLL.PLLR = RCC_PLLR_DIV2;
if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
{
 Error_Handler();
}

RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
 |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_4) != HAL_OK)
{
 Error_Handler();
}

if (HAL_PWREx_ControlVoltageScaling(PWR_REGULATOR_VOLTAGE_SCALE1) != HAL_OK)
{
 Error_Handler();
}
}

static void MX_SPI1_Init(void)
{
 hspi1.Instance = SPI1;
 hspi1.Init.Mode = SPI_MODE_MASTER;
 hspi1.Init.Direction = SPI_DIRECTION_2LINES;
 hspi1.Init.DataSize = SPI_DATASIZE_8BIT;
 hspi1.Init.CLKPolarity = SPI_POLARITY_LOW;
 hspi1.Init.CLKPhase = SPI_PHASE_1EDGE;
 hspi1.Init.NSS = SPI_NSS_SOFT;
```

```

 hspi1.Init.BaudRatePrescaler = SPI_BAUDRATEPRESCALER_2;
 hspi1.Init.FirstBit = SPI_FIRSTBIT_MSB;
 hspi1.Init.TIMode = SPI_TIMODE_DISABLE;
 hspi1.Init.CRCCalculation = SPI_CRCCALCULATION_DISABLE;
 hspi1.Init.CRCPolynomial = 7;
 hspi1.Init.CRCLength = SPI_CRC_LENGTH_DATASIZE;
 hspi1.Init.NSSPMode = SPI_NSS_PULSE_DISABLE;
 if (HAL_SPI_Init(&hspi1) != HAL_OK)
 {
 Error_Handler();
 }
}

static void MX_GPIO_Init(void)
{
 GPIO_InitTypeDef GPIO_InitStruct = {0};

 /* GPIO Ports Clock Enable */
 __HAL_RCC_GPIOC_CLK_ENABLE();
 __HAL_RCC_GPIOA_CLK_ENABLE();

 /*Configure GPIO pin Output Level */
 HAL_GPIO_WritePin(GPIOC, GPIO_PIN_0, GPIO_PIN_RESET);

 /*Configure GPIO pin : PC0 */
 GPIO_InitStruct.Pin = GPIO_PIN_0;
 GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
 GPIO_InitStruct.Pull = GPIO_NOPULL;
 GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
 HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);
}

void Error_Handler(void)
{
}

#ifdef USE_FULL_ASSERT

void assert_failed(uint8_t *file, uint32_t line)
{
}

#endif

```

```

/***** (C) COPYRIGHT STMicroelectronics *****/

```

*Figure 13.9: The SPILED program.*

The following HAL functions are available for the SPI protocol:

**to transmit data to a slave device:**

```

HAL_SPI_Transmit(handle, uint8_t *pData, uint16_t Size, uint32_t Timeout);

```

**to read data from a slave device:**

```

HAL_SPI_Receive(handle, uint8_t *pData, uint16_t Size, uint32_t Timeout);

```

**to transmit and receive in full-duplex mode:**

```

HAL_SPI_TransmitReceive(handle, uint8_t *pTxData, uint8_t *pRxData, uint16_t
Size, uint32_t Timeout);

```

**to transmit in interrupt mode:**

```

HAL_SPI_Transmit_IT(handle, uint8_t *pData, uint16_t Size);

```

**to receive in interrupt mode:**

```

HAL_SPI_Receive_IT(handle, uint8_t *pData, uint16_t Size);

```

**to transmit and receive in interrupt mode:**

```

HAL_SPI_TransmitReceive_IT(handle, uint8_t *pTxData, uint8_t *pRxData,
uint16_t Size);

```

**The callback functions available in interrupt mode are:**

|                                |                                                                          |
|--------------------------------|--------------------------------------------------------------------------|
| HAL_SPI_RxCpltCallback()       | Specified no. of bytes have been received                                |
| HAL_SPI_RxHalfCpltCallback()   | Half of the specified no. of bytes received                              |
| HAL_SPI_TxCpltCallback()       | Specified no of bytes have been transmitted                              |
| HAL_SPI_TxHalfCpltCallback()   | Half of the specified no. of bytes have been transmitted                 |
| HAL_SPI_TxRxCpltCallback()     | Specified no. of bytes have been transmitted and re-<br>ceived           |
| HAL_SPI_TxRxHalfCpltCallback() | Half of the specified no. of bytes have been transmitted<br>and received |

### 13.4 Summary

In this Chapter we have actively developed a project based on the SPI bus. Additionally, a list of the SPI HAL functions has been discussed. In the next Chapter we will be looking at debugging our programs.

## CHAPTER 14 • Program Debugging

### 14.1 Overview

In last Chapter we have learned how to interface the SPI devices to our development board and also how to program the STM32CubeIDE software for the SPI bus. In this Chapter we will be learning how to debug programs.

Debugging is the process of detecting and removing existing errors (also called ‘bugs’) in a software code. Debugging is used to prevent incorrect operation of software, or to find out why a software code is not doing what it is supposed to do. As the complexity of software grows, it becomes more difficult to test and find any present or potential errors. Sometimes it may take longer time to debug a program than to code it.

There are many software debugging tools that can be used to identify coding errors. For example, programmers can trace the program execution step-by-step and observe the values of the variables at different program steps. Values of variables can be changed during debugging and the behaviour of the program can be examined. It is important to realize that debugging is not something that can magically reveal the problems in our code. We have to run the code step-by-step in order to find out where the problem might be.

Most integrated development environment based compilers offer debugging tools in the form of software and hardware. For example, using a hardware debugger a programmer can step through the code on the actual microcontroller in real time.

The ST-LINK is the JTAG/Serial Wire Debug (SWD) interface (Figure 14.1) is needed to communicate with any STM32 microcontroller located on an application board so that the code can be debugged. Fortunately, all STM32 Nucleo boards integrate an ST-LINK debugger/programmer, so there is no need for a separate probe. The ST-LINK debugger/programmer is located in the upper part of the Nucleo boards.



*Figure 14.1: T-LINK, ST-LINK/V2, and ST-LINK/V2-ISOL standalone probes.*

In this Chapter we will create a very small program and then see how we can debug the code.

### 14.2 Project 1: Simple Debug

As debugging example, we will create a very simple program that will increment the value of a variable from 0 to 10 with one second delay between each count.

On STMicroelectronics hardware kits, SWD must be made available for connection with ST-LINK. SWD is always mapped on PA13 (SWDIO) and PA14 (SWCLK). This is the default state after reset.

The steps are as follows.

- Start the STM32CubeIDE.
- Create a new workspace.
- Name the program as **DEBUGLRN**.
- Click **SYS** under **System Core** and set **Debug** to **Serial Wire** (Figure 14.2). You will notice that PA13 and PA14 will be configured as SYS\_JTMS-SWDIO and SYS\_JTICK respectively.

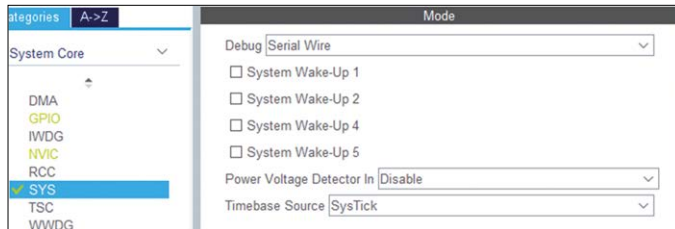


Figure 14.2: Enabling Debug.

- Configure the MCU clock for 80 MHz.
- Click **File**, followed by **Save**, and click **YES** to generate code.
- Click **Core**, followed by **Src**, and double click **main.c** to open the main program.

Compilers are usually configured by default to optimize performance and/or code size. In most cases, this reduces or even prevents program debugging. It is therefore necessary to disable compiler optimization. This is done as follows (Figure 14.3).

- Click **Project**, followed by **Properties**. Click to expand tab **C/C++ Build**.
- Click **Settings**. Click tab **Tool Settings**. Click **Optimization** under **MCU GCC Compiler**.
- Set the **Optimization level** is set to **None (-O0)**.
- Click **Apply and Close**.

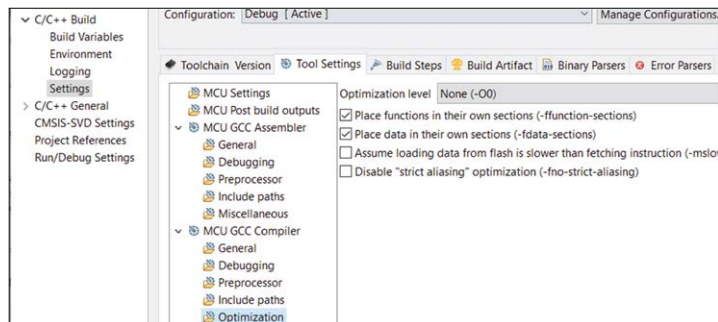


Figure 14.3: Removing optimization.

Debugging information is generated by the compiler together with the machine code. It is a representation of the relationship between the executable program and the original source code. This information is encoded into a pre-defined format and stored alongside the ma-



chine code. Debugging information is mandatory to set breakpoint or get the content of a variable. The steps to enable the debugging information are as follows.

- Click **Project**, followed by **Properties**. Click to expand tab **C/C++Build**.
- Click **Settings**. Click tab **Tool Settings**.
- Click **Debugging** under **MCU GCC Compiler**.
- Set the **Debug level** to **Maximum (-g3)**.
- Click **Apply and Close**.

We can now write our code inside the main program as follows (the clock code, error code etc are not shown).

```
#include "main.h"
void SystemClock_Config(void);
static void MX_GPIO_Init(void);

int main(void)
{
 int count = 0;

 HAL_Init();
 SystemClock_Config();
 MX_GPIO_Init();

 while (1)
 {
 count++;
 if(count == 10)count = 0;
 HAL_Delay(1000);
 }
}
```

- Now click **Project**, followed by **Build Configurations**, and then click **Set Active**. Make sure that it is set to **Debug**.
- Click **Project**, followed by **Build All** to compile the program in debug mode. Make sure there are no compilation errors.
- Make sure that your Nucleo board is plugged into the USB port of your PC.
- Click **Run**, followed by **Debug As** and click **STM32 Cortex-M C/C++ Application**
- Click **Switch** to accept application debugging.

You should now see the debug screen (Figure 14.4) displayed with the program in the middle pane.

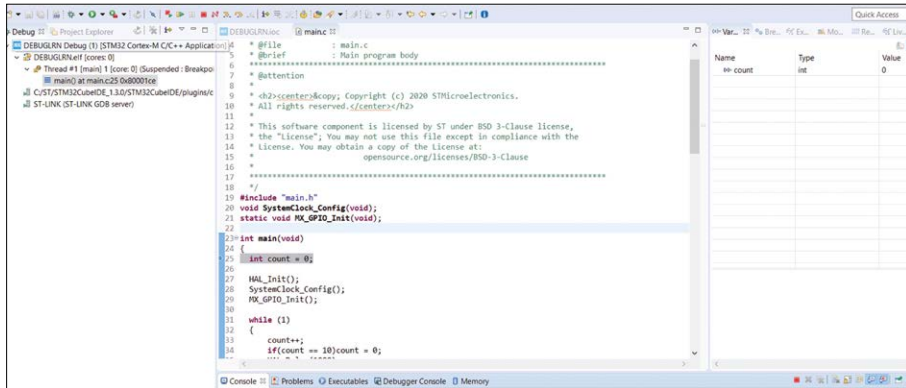


Figure 14.4: The Debug screen.

Click **Run** to see the various debug options. Some useful options are (these options are also available as icons at the top of the screen):

- F5            Step into
- F6            Step over
- Cntrl+R    Run to line
- F8            Resume
- Cntrl+F2    Terminate

Let's now single step through the program and watch the value of variable **count**.

- The current cursor position is highlighted, and this is where the program counter is at the beginning of the program.
- Press **F6** to step through the program. Keep pressing **F6** and you should see the program statements being highlighted as the program executes each statement.
- After executing the line containing statement **count++**, you should see variable **count** incremented to 1 at the top right pane of the screen. Make sure **Var...** (i.e. **Variables**) tab is selected (Figure 14.5).

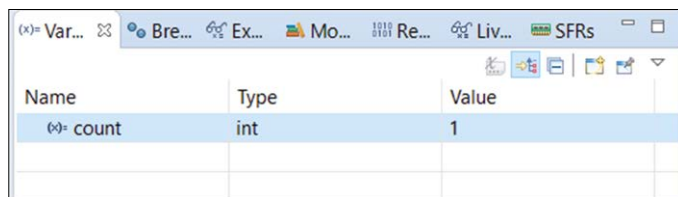


Figure 14.5: Variable 'count' is incremented.

- You should see variable **count** incremented every time the statement **count++** is executed.
- Place the cursor over variable **count** in the program and you should see its value displayed in a pop-up window.
- Now click **Run**, followed by **Terminate and Relaunch** (or click **Reset**).

Now let's put a breakpoint at the statement `HAL_Delay(1000)` so that the program stops here when run without single stepping.

- Put the cursor to the left of statement `HAL_Delay(1000)` and double click in the blue line at the left hand side. You should see a little mark placed next to the statement.
- Now, click **Run**, followed by **Run to Line**. You should see the program running and stopping at the statement `HAL_Delay(1000)` with this statement highlighted and **count** incremented. Click **Run**, followed by **Run to Line** again and the program will run and stop at `HAL_Delay(1000)`. Variable **count** will be incremented again.
- To remove the breakpoint, place the cursor over the breakpoint mark and right click the mouse to select **Toggle Breakpoint**.
- To change the value of variable **count**, click **Value** at the right-hand side where **count** is displayed and enter a new value.
- Click **Run**, followed by **Terminate** to terminate the debug session (or click the red square icon).

### 14.3 Project 2: Debugging the GPIO

In this example we will see how to debug a very simple program and monitor the state of a GPIO port. The program flashes the on-board LED connected to port PA5 every 100 ms. The program listing is given below, where only the main program is shown.

```
include "main.h"
#define LED GPIO_PIN_5

void SystemClock_Config(void);
static void MX_GPIO_Init(void);

int main(void)
{
 HAL_Init();
 SystemClock_Config();
 MX_GPIO_Init();

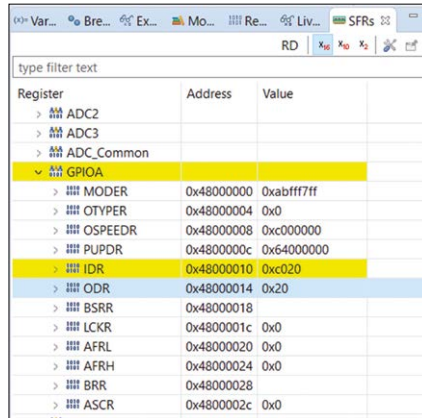
 while (1)
 {
 HAL_GPIO_WritePin(GPIOA, LED, GPIO_PIN_SET);
 HAL_Delay(100);
 HAL_GPIO_WritePin(GPIOA, LED, GPIO_PIN_RESET);
 HAL_Delay(100);
 }
}
```

Compile the program in **Debug** mode as before and start the debugger. Single step through the program by pressing F6, and observe that the LED turns ON and OFF inside the program loop as you step through the code.

To see the state of port GPIOA, click tab **SFRs** at the top right hand side, and select GPIOA. Step through the program and stop after the statement to turn ON the LED. You should see the output register ODR of GPIOA set to 0x20 which corresponds to binary (Figure 14.6):

0000 0000 0010 0000

Where bit 5 (i.e. port pin PA5) is set to turn ON the LED. You can change the ODR to 0x00 to turn OFF the LED. Click the red square icon to terminate debugger.



| Register     | Address    | Value      |
|--------------|------------|------------|
| > ADC2       |            |            |
| > ADC3       |            |            |
| > ADC_Common |            |            |
| > GPIOA      |            |            |
| > MODER      | 0x48000000 | 0xabfff7ff |
| > OTYPER     | 0x48000004 | 0x0        |
| > OSPEEDR    | 0x48000008 | 0xc0000000 |
| > PUPDR      | 0x4800000c | 0x64000000 |
| > IDR        | 0x48000010 | 0xc020     |
| > ODR        | 0x48000014 | 0x20       |
| > BSRR       | 0x48000018 |            |
| > LCKR       | 0x4800001c | 0x0        |
| > AFR1       | 0x48000020 | 0x0        |
| > AFRH       | 0x48000024 | 0x0        |
| > BRR        | 0x48000028 |            |
| > ASCR       | 0x4800002c | 0x0        |

Figure 14.6: State of the GPIOA registers.

#### 14.4 Project 3: Displaying Characters in Debug Window

It's very useful to display characters at various points in a program while it is being debugged. In this example we will see how to display single characters in the STM32CubeIDE's debug window.

The main program listing is given below. The program displays characters **A** and **B** in the debug window using function call **ITM\_SendChar**:

```
#include "main.h"

void SystemClock_Config(void);
static void MX_GPIO_Init(void);

int main(void)
{
 HAL_Init();
 SystemClock_Config();
 MX_GPIO_Init();

 ITM_SendChar('A');
 ITM_SendChar('B');

 while (1)
```

```

{
}
}

```

Click tab **System Core**, followed by **SYS** and set **Debug** to **Serial Wire** (Figure 14.7). Click **File**, followed by **Save** and click **YES** to generate code.

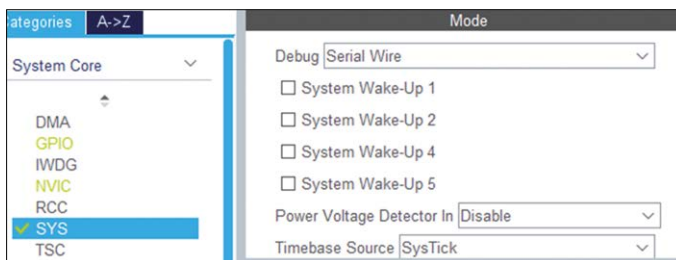


Figure 14.7: Set debug to Serial Wire.

Open the main program. Compile the program in **Debug** mode. Before debugging the program, we must configure various windows as described below.

- Click **Run**, followed by **Debug Configurations**.
- Click tab **Debugger**.
- Set **Interface** to **SWD**, Enable **Serial Wire Viewer (SWV)**, set **Core Clock** to the clock frequency selected for the MCU (e.g. 80 MHz), and click **Close** (see Figure 14.8).

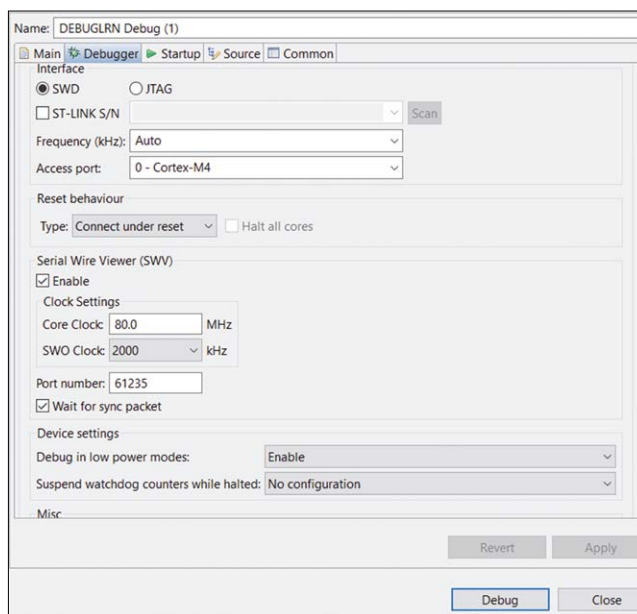


Figure 14.8: Configuring the Debug Configurations.

- Start the debug session by clicking **Run**, followed by **Debug As** and click **STM32 Cortex-M C/C++ Application**.
- Click **Window**, followed by **Show View**, then **SWV**, followed by **SWV ITM Data Console**. You should see the **SWV ITM Data Console** at the bottom of the screen.
- Click on tab **SWV ITM Data Console** to select it. Click **Configure trace** button to configure **SWV ITM Data Console** (Figure 14.9).



Figure 14.9: Click for SWV settings.

- Click to Enable Port 0 (Figure 14.10), click **OK**.

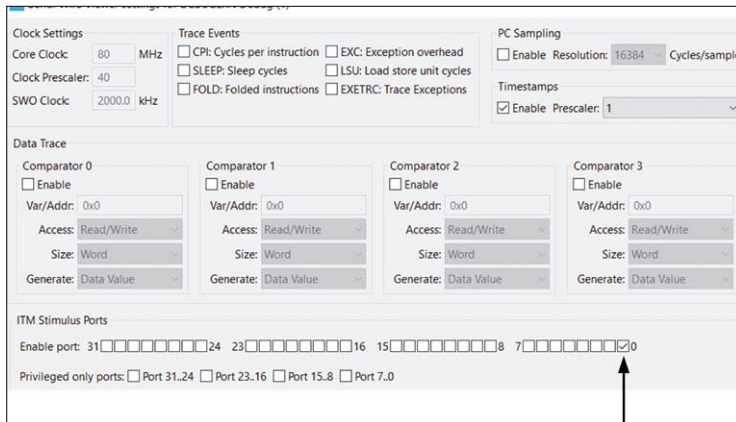


Figure 14.10: Enabling Port 0.

- Click **Start Trace** (little red circle in Figure 14.11).



Figure 14.11: Click: Start Trace.

- We have now completed the configurations. Single step your program and you should see characters **AB** displayed in the debug window (Figure 14.12).

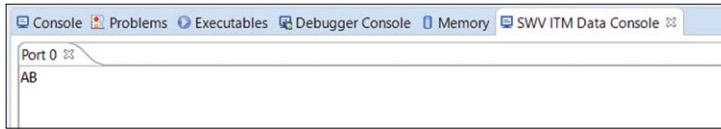


Figure 14.12: Displaying data in the debug window.

### 14.5 Project 4: Using 'printf' to Display Data in Debug Window

In previous example we have seen how to display characters in the debug window. In this example we will see how to use the printf statement to display text as well as numeric data in the debug window using the well-known printf statement.

The configuration steps given in the previous example must be applied for this example as well. Additionally, we have to create a new function called e.g. **my\_printf** with the contents as follows. Insert this function to your program before the main program (the clock, error routines, and comments are not shown here for clarity):

```
#include "main.h"
#include "stdio.h"
#include "stdarg.h"

void SystemClock_Config(void);
static void MX_GPIO_Init(void);

void my_printf(const char *fmt, ...)
{
 char buffer[256];
 va_list args;
 va_start(args, fmt);
 vsnprintf(buffer, sizeof(buffer), fmt, args);
 va_end(args);

 uint16_t i = 0;
 while(buffer[i] != '\0')
 {
 ITM_SendChar(buffer[i]);
 i++;
 }
}

int main(void)
{
 int count = 10;
 float fl = 25.21;
 HAL_Init();
 SystemClock_Config();
 MX_GPIO_Init();
```

```

my_printf("Hello from the debugger...\n");
my_printf("count=%d fl=%5.2f", count, fl);

while (1)
{
}
}

```

Notice that in order to display floating point numbers you will have to modify the linker settings before running the debugger. The steps are given here.

- Click **Project**, followed by **Properties**.
- Expand tab **C/C++ Build** and click **Settings**.
- Click tab **Tool Settings** and click **Miscellaneous** under tab **MCU GCC Linker**.
- Click icon **Add** (the first icon under **Other flags**).
- Enter string **-u\_printf\_float** as shown in Figure 14.13.
- Click **Apply and Close**.

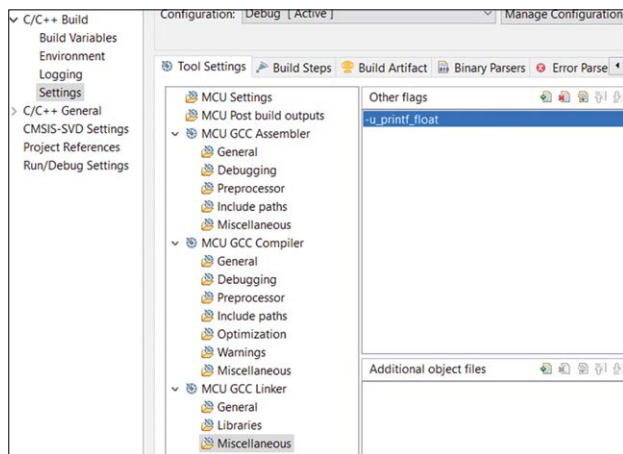


Figure 14.13: Configuring the linker for floating-point.

Single step (or set a breakpoint at the **while** statement and **Run to Line**) the program and the program counter reached the while statement you should see the outputs of the printf statements displayed on the debug console as shown in Figure 14.14.

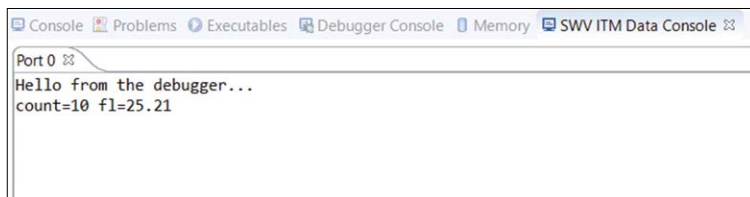


Figure 14.14: The Debug window.



### 14.6 Project 5: Using the ST-Link Virtual COM Port

We can also run a terminal emulator software on the PC (e.g. HyperTrm, Putty, etc.) and configure our Nucleo board to use USART2. As shown in the Chapter 11 on using the serial line, we can insert statements at various points in our program to send messages (using function **HAL\_UART\_Transmit**) to the PC in order to debug our program while it is running in **Release** mode. Using the UART is not discussed here any further, since it is covered in detail in Chapter 11.

### 14.7 Summary

In this Chapter we have learned how to debug our program using the STM32CubeIDE debugger. In next Chapter we will focus on the important topic of power management while using our Nucleo development board.

## CHAPTER 15 • STM32L4 MCU Power Management

### 15.1 Overview

The STM32L series (L0, L1, and L4) has been developed for low-power applications. In this Chapter we will be focusing on the STM32L4 series power management features.  $V_{DD}$  is the main power supply for the MCU. When  $V_{DD}$  is turned OFF, an optional battery connected to VBAT pin can be used to provide standby voltage. The battery powers the Real-Time Clock (RTC) module, the LSE oscillator, and wake-up pins to wake up the MCU from deep sleep modes. When  $V_{DD}$  is present, it is possible to charge the external battery through an internal resistor.

The MCU is placed in sleep mode using instructions WFI (Wait for Interrupt) and WFE (Wait for Event). When in sleep mode, the CPU will be resumed by an interrupt request. The WFE is similar to WFI, but it checks the status of an event register and puts the CPU into sleep mode if the event is not already set. The WFE can be woken up by external events.

### 15.2 Low power modes

The STM32L4 MCU low-power modes are depicted in Figure 15.1.

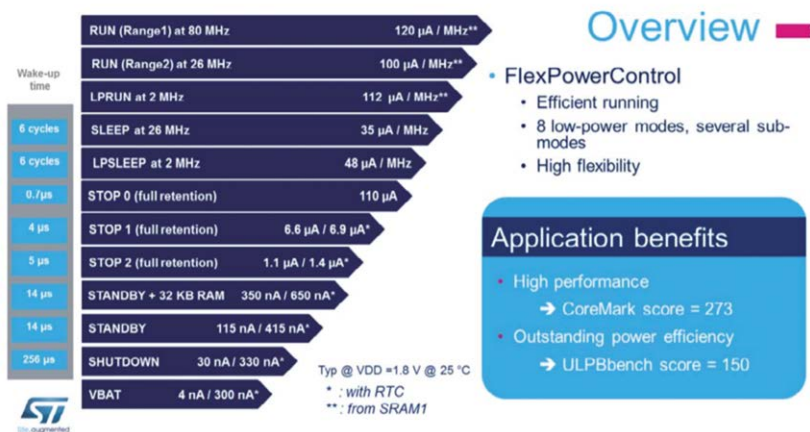


Figure 15.1: STM32L4 MCU low-power modes.

**Run mode:** two run modes are available: Range 1 and Range 2. Figure 15.2 shows the system overview at Range 1. This is the default mode after power-on or reset, where the clock runs at full speed of 80 MHz, consuming 131  $\mu$ A/MHz, all peripherals can be activated, all clocks can be enabled, the flash memory is enabled.

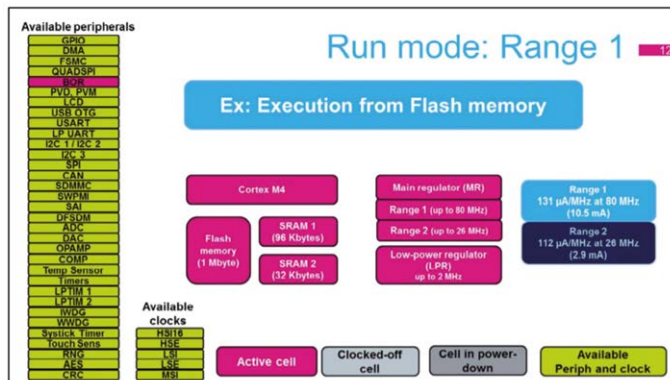


Figure 15.2: Run mode Range 1.

Run mode Range 2 (Figure 15.3) is a medium-performance range with the clock up to 26 MHz, consuming 110 µA/MHz with execution from SRAM, and all clocks can be enabled. When executing from SRAM, the Flash power consumption is saved. All peripherals can be activated except the USB OTG and Random Number Generator.

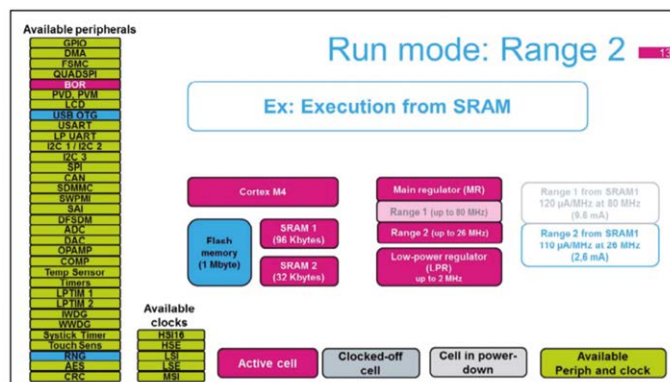


Figure 15.3: Run mode Range 2.

The flash memory can be disabled in the run mode by calling this function:

```
HAL_FLASHEx_EnableRunPowerDown()
```

And re-enabled by calling this function:

```
HAL_FLASHEx_DisableRunPowerDown()
```

It is a requirement that all routines be placed in SRAM when the flash is OFF, otherwise bus faults will occur.

Run Range 2 is entered by calling this function:

```
HAL_PWREx_EnableVoltageScaling()
```

**Low-power run mode:** in this mode (Figure 15.4) the main regulator is OFF. The system clock is up to 2 MHz, all clocks can be enabled. Execution can be from Flash memory or SRAM. The Flash consumption can be reduced by powering it down and gating its clock off. All peripherals can be activated except the USB OTG and Random Number Generator. Power consumption is 135  $\mu\text{A}/\text{MHz}$  when executing from Flash memory, and 112  $\mu\text{A}/\text{MHz}$  when operating from SRAM.

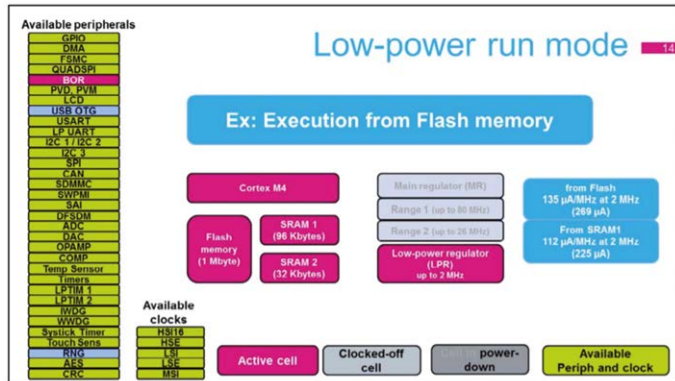


Figure 15.4: Low-power run mode.

This mode is entered by calling:

```
HAL_PWREx_EnableLowPowerRunMode()
```

**Sleep mode:** in this mode (Figure 15.5), the CPU clock is off, system clock can be up to 80MHz. All peripherals can be activated. SRAM clocks are enabled but can be gated off. Power consumption is 37  $\mu\text{A}/\text{MHz}$ .

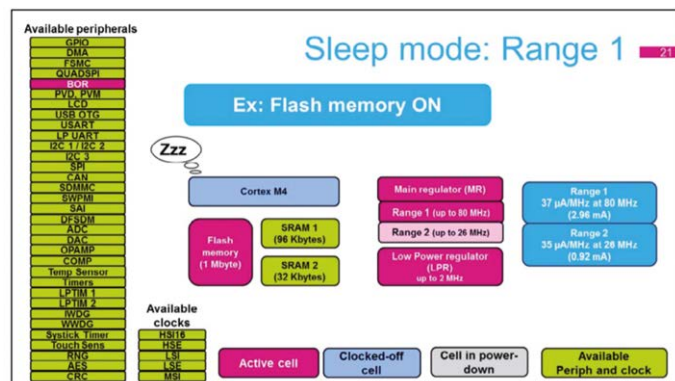


Figure 15.5: Sleep mode.

This mode is entered by calling function:

```
HAL_PWR_EnterSLEEPMode()
```

The CPU wakes up from the Sleep condition by an interrupt (WFI or WFE).

**Low-power sleep mode:** in this mode (Figure 15.6) the CPU clock is OFF, the system clock is reduced to 2 MHz. Flash memory can be gated off. All peripherals can be activated except the USB OTG and Random Number Generator. the power consumption is 40  $\mu\text{A}/\text{MHz}$  at 2 MHz with Flash and SRAM OFF.

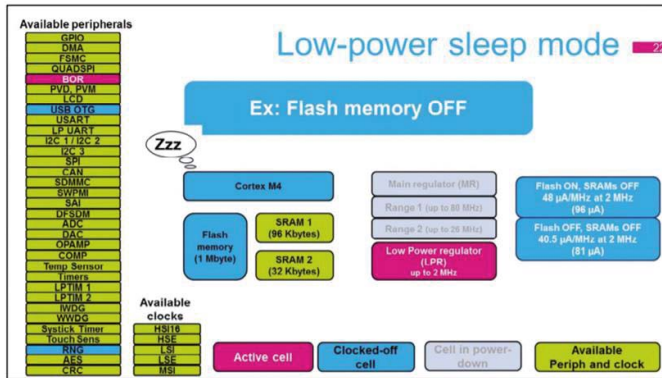


Figure 15.6: Low-power sleep mode.

This mode is entered by calling function:

```
HAL_PWR_EnterSLEEPMode()
```

**Stop 0, Stop 1 and Stop 2 modes:** these are the lowest power modes. The contents of SRAM and all peripherals registers are preserved. All high-speed clocks are stopped, but the 32,768 kHz external and 32 kHz internal clocks can be enabled. The system clock on wake-up can be the internal high-speed clock up to 48 MHz with 0.7  $\mu\text{s}$  wake-up time from SRAM, or 5  $\mu\text{s}$  from Flash. Stop-2 power consumption is lower than Stop-1 and Stop-0. The CPU exits from the Stop condition by any EXTI interrupt or by reception of interrupts from UART, I<sup>2</sup>C, etc.

Figure 15.7 shows the Stop mode 0. In this mode, the system clock is frozen, most peripheral clocks are gated off. Several peripherals can be functional: Power voltage detector, peripherals voltage monitor, LCD controller, digital to analogue converters, operational amplifiers, comparators, independent watchdog, low power timers, I<sup>2</sup>C, UART and low-power UART. The events from all I/Os can wake up from Stop 0 mode, plus the interrupt generated by the active peripherals.

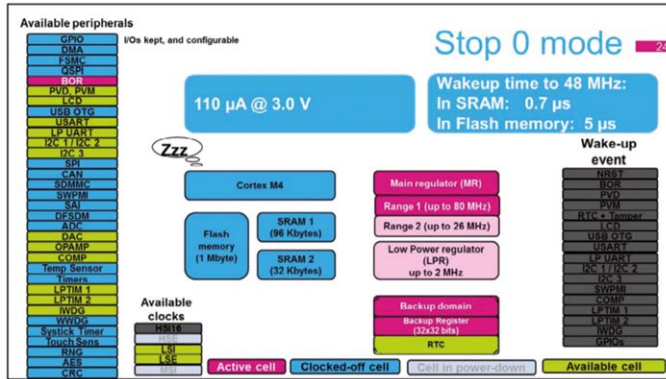


Figure 15.7: Stop mode 0.

**Stop 1 mode** is similar to Stop 0 mode, except that the power consumption is lower. Stop 1 mode is entered by calling function:

```
HAL_PWREx_EnterSTOP1Mode()
```

**Stop 2 mode** is similar to Stop 1 mode. Several peripherals can be functional: power voltage detector, peripheral voltage monitors, LCD controller, comparators, independent watchdog, low power timer 1, I2C3, and the low-power UART. The events from all I/Os can wake up from Stop 2 mode, plus the interrupt generated by the active peripherals. Stop 1 mode is entered by calling function:

```
HAL_PWREx_EnterSTOP2Mode()
```

**Standby mode:** this is low- power mode (Figure 15.8). The 32 Kbytes of SRAM2 can be retained. Voltage regulators are in power down mode, SRAM and peripherals registers are lost. 128-byte backup registers are retained. The CPU exits from the Standby condition by WKUP pin rising edge, RTC alarm, external reset, or IWDG reset.

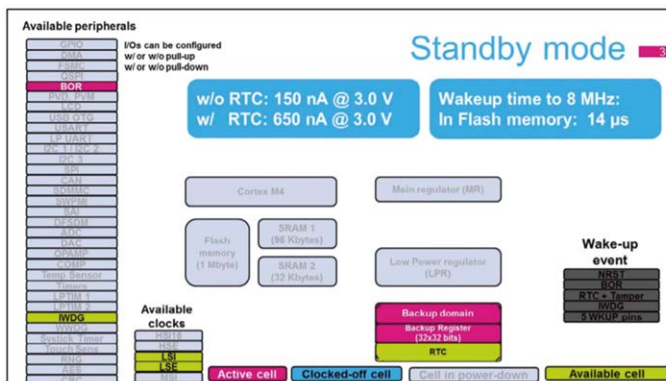


Figure 15.8: Standby mode.

Standby mode is entered by calling function:

```
HAL_PWR_EnterSTANDBYMode()
```

**Shutdown mode:** this is the lowest power mode (Figure 15.9) with only 30 nA at 1.8 V. This mode is similar to Standby mode but without any power monitoring: the brown-out reset is disabled and the switch to VBAT is not supported in Shutdown mode. The 128-byte backup registers are retained in Shutdown mode. The CPU exits from the Shutdown mode by WKUP pin rising edge, RTC alarm, or external reset.

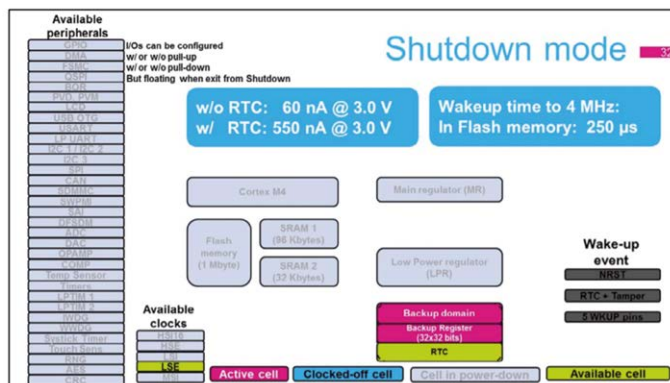


Figure 15.9: Shutdown mode.

Shutdown mode is entered by calling function:

```
HAL_PWREx_EnterSHUTDOWNMode()
```

### 15.3 Power modes transitions

Figure 15.10 shows the possible transitions from one power mode to another one. It is clear from the figure that from Run mode we can access all low-power modes except the Low-power sleep mode. Before moving to Low-power sleep mode, first we have to move to Low-power run mode and execute a WFI or WFE instruction. When exiting the Low-power sleep mode we always return to the Low-power run mode. If the processor is in the Standby or Shutdown modes, then it will always return to the Run mode.

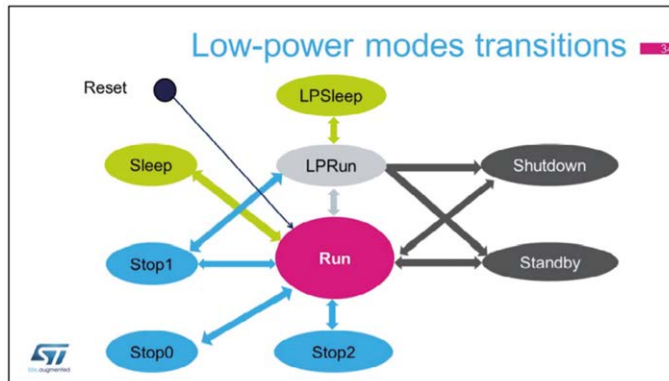


Figure 15.10: Power mode transitions.

### 15.4 Low power peripherals

The STM32L4 processor support low-power peripherals such as low-power UART (LPUART), and low-power timer (LPTIM).

### 15.5 Debugging in low-power modes

If the MCU is placed into sleep, stop, or standby modes while debugging then the debug connection is lost. We can however call some HAL functions to enable debugging in low-power modes. For example, function call **HAL\_DBGMCU\_EnableDBGSleepMode()** enables debugging in sleep mode. Also, **HAL\_DBGMCU\_EnableDBGStopMode()** enables debugging in stop mode. Because debugging uses the GPIO pins PA13, PA14 and PC3, we have to enable the GPIOA and GPIOC during debugging.

### 15.6 Measuring Nucleo current consumption

The Nucleo boards include IDD pin headers that can be used to measure the MCU current consumption. The IDD jumper should be removed and an ammeter should be connected between these pins to measure the MCU current consumption at any time.

### 15.7 Project 1: Sleep Mode Example

#### Description

In this project we will be entering the sleep mode. The on-board button (at PC13, normally HIGH) and the on-board LED (at PA5) are used in this program. Remember that the button state is normally logic HIGH and goes to logic LOW when the button is pressed. In this project, the MCU will wake up whenever an external interrupt is detected by pressing the button. The state of the on-board LED will then be toggled inside the ISR. The sleep on exit mode will be set so that the program returns to sleep mode after exiting the ISR. As a result, any code outside the ISR will not be executed.

#### The aim

The aim of this project is to show how the MCU can be put into Sleep mode and also how it can exit this mode.



## Program listing

We will enter the sleep mode by executing either WFI or WFE instructions. Notice that using the Wait For Interrupt (WFI) will wake up the MCU when any peripheral interrupt is acknowledged. In this project we will use the WFI. The function call to enable WFI with the main regulator enabled is:

```
HAL_PWR_EnterSLEEPMode(PWR_MAINREGULATOR_ON, PWR_SLEEPENTRY_WFI);
```

The MCU will wake up whenever an external interrupt is triggered (i.e. when the button is pressed) and the ISR is the following callback function. Inside the callback function we will toggle the LED.

```
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
 // Toggle the LED
}
```

We will be enabling the sleep-on-exit feature so that when the interrupt occurs it will be processed inside the ISR, and the MCU will go back into sleep mode when the ISR exits. The function call to enable sleep-on-exit is:

```
HAL_PWR_EnableSleepOnExit();
```

In some applications we may have to suspend ticks by disabling systick interrupts. This is done by calling the following function:

```
HAL_SuspendTick();
```

To resume systick interrupts, use the following function:

```
HAL_ResumeTick();
```

The steps to develop the program are as follows.

- Start the STM32CubeIDE.
- Create a new workspace.
- Start a new SMT32 project and select **STM32L476RG** as the processor.
- Name the program as **SLEEP**.
- Configure **PA5** as digital output and **PC13** as **GPIO\_EXTI13**. Right click the mouse and set the label of PA5 as **LED** and the label of PC13 as **Button** (Figure 15.11).

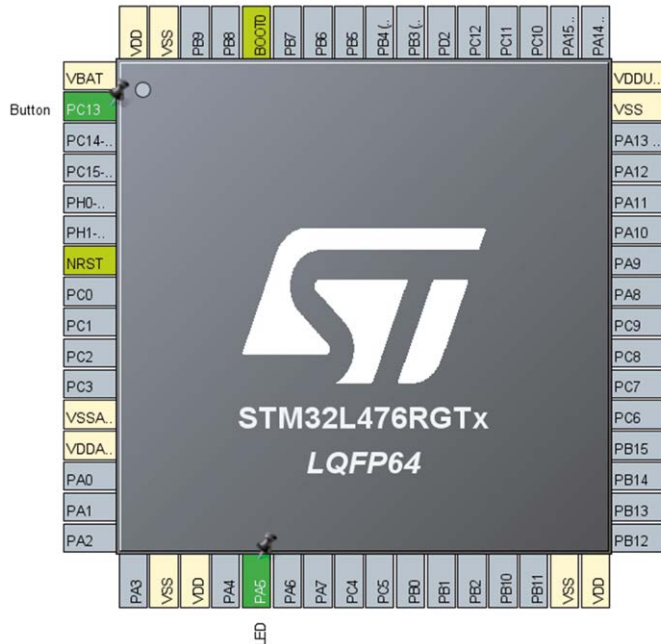


Figure 15.11: Configure PA5 and PC13.

- Click **System Core** and select **GPIO**. Click **PC13** and set the GPIO mode to **External Interrupt Mode with Falling edge trigger detect**.
- Click **NVIC** under tab **System Core**. Click to enable **EXTI line[15:10] interrupts**
- Click **File**, followed by **Save** and click **YES** to generate code.
- Click **Core**, followed by **Src** and double click **main.c** to open the main program.

The ISR code toggles the LED, and its code is as follows:

```
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
 HAL_GPIO_TogglePin(GPIOA, LED_Pin);
}
```

Inside the main program, sleep-on-exit is enabled and then the CPU is put into sleep mode by calling the following functions:

```
HAL_PWR_EnableSleepOnExit();
HAL_PWR_EnterSLEEPMode(PWR_MAINREGULATOR_ON, PWR_SLEEPENTRY_WFI);
```

Notice that the following code is placed inside the program loop, but these statements will not be executed since the CPU goes back to sleep after the ISR.

```
//
// The following code will not be executed since we have enabled sleep-on-exit
//
while (1)
{
 HAL_GPIO_WritePin(GPIOA,LED_Pin, GPIO_PIN_SET);
 HAL_Delay(500);
 HAL_GPIO_WritePin(GPIOA,LED_Pin, GPIO_PIN_RESET);
 HAL_Delay(500);
}
```

If we insert the statement **HAL\_PWR\_DiableSleepOnExit()** inside the ISR, then when the button is pressed the LED will toggle its state and then the statements inside the program loop will be executed, flashing the LED every 500 ms.

Figure 15.12 shows the program listing (Program: **SLEEP**), where the clock, I/O, and error routines are not shown for clarity.

```
#include «main.h»

void SystemClock_Config(void);
static void MX_GPIO_Init(void);

//
// This is the interrupt service routine (ISR). The program jumps here when the
// Button is pressed. Here, we toggle the LED
//
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
 HAL_GPIO_TogglePin(GPIOA, LED_Pin);
}

//
// Start of main program
//
int main(void)
{
 HAL_Init();
 SystemClock_Config();
 MX_GPIO_Init();

 HAL_PWR_EnableSleepOnExit();
 HAL_PWR_EnterSLEEPMode(PWR_MAINREGULATOR_ON, PWR_SLEEPENTRY_WFI);

//
// The following code will not be executed since we have enabled sleep-on-exit
```

```
//
while (1)
{
 HAL_GPIO_WritePin(GPIOA,LED_Pin, GPIO_PIN_SET);
 HAL_Delay(500);
 HAL_GPIO_WritePin(GPIOA,LED_Pin, GPIO_PIN_RESET);
 HAL_Delay(500);
}
}
```

*Figure 15.12: The SLEEP Program.*

## 15.8 Project 2: Stop Mode Example

### Description

This project is similar to the previous one where the on-board button and the on-board LED are used.

### The aim

The aim of this project is to show how the MCU Stop mode can be used in an example.

### Program listing

In Stop mode, all clocks are stopped, PLLs, the HSI, and the HSE RC oscillator are disabled. Internal MCU registers and SRAM contents are preserved. **In this project we will be turning OFF the main Regulator and using the Low Power Regulator.**

We can enter the stop mode by executing WFI or WFE as in the sleep mode. If the WFI is used to enter the stop mode, an external interrupt, watchdog, or RTC can wake up the device. In this example, external interrupt generated by pressing the on-board button will be used.

The following function will be used to enter the stop mode:

```
HAL_PWR_EnterSTOPMode(PWR_LOWPOWERREGULATOR_ON, PWR_STOPENTRY_WFI);
```

Because all clocks are stopped in this mode, we have to restart the system clock inside the ISR. It may also be required to resume the systick in some applications:

```
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
 if(GPIO_Pin == GPIO_PIN_13)
 {
 SystemClock_Config ();
 }
}
```

Again, we will be enabling the sleep-on-exit so that the CPU returns to sleep mode after exiting the ISR. The program of this project is very similar to the one given here and as a result, the program development steps are not repeated here.

The program listing is shown in Figure 15.13. Again, only the main program listing is shown. Notice that the LED flashing code inside the program loop is not executed since the sleep-on-exit is enabled. If we disable the sleep-on-exit inside the ISR then we will see the LED flashing every 500 ms after the button is pressed once. This requires the CPU clock to be running, which is the case here since the clock is configured inside the ISR.

```
#include «main.h»

void SystemClock_Config(void);
static void MX_GPIO_Init(void);

//
// This is the interrupt service routine (ISR). The program jumps here when the
// Button is pressed. Here, we toggle the LED
//
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
 SystemClock_Config ();
 HAL_GPIO_TogglePin(GPIOA, LED_Pin);
}

//
// Start of main program
//
int main(void)
{
 HAL_Init();
 SystemClock_Config();
 MX_GPIO_Init();

 HAL_PWR_EnableSleepOnExit();
 HAL_PWR_EnterSTOPMode(PWR_LOWPOWERREGULATOR_ON, PWR_STOPENTRY_WFI);

 //
 // The following code will not be executed since we have enabled sleep-on-exit
 //
 while (1)
 {
 HAL_GPIO_WritePin(GPIOA, LED_Pin, GPIO_PIN_SET);
 HAL_Delay(500);
 HAL_GPIO_WritePin(GPIOA, LED_Pin, GPIO_PIN_RESET);
 HAL_Delay(500);
 }
}
```

*Figure 15.13: Program listing.*

### 15.9 Project 3: Standby Mode Example

#### Description

This project is similar to the previous one. Here, the on-board LED is toggled when the CPU wakes up from Standby mode. The wakeup occurs if the wakeup pin (GPIO pin PA0) rises from LOW to HIGH.

#### The aim

The aim of this project is to show how the MCU Standby mode can be used in an example.

#### Program listing

In this mode very low power is consumed. The PLLs, the HSI and the HSE oscillator are all cut-off.

The Standby mode is virtually same as the system Reset where after a wakeup the program runs from the beginning as if a Reset occurred. The only difference between the Reset and the standby wakeup is that in Standby wakeup the CPU flag **PWR\_FLAG\_SB** is set. Thus, by checking the value of this flag we can determine whether or not a Reset or a wakeup from Standby mode has occurred.

The standby wakeup can be triggered on the rising edge of the **WKUP 1** pin (GPIO pin PA0 on STM32L476RG microcontroller), or it can wake up by the RTC, or IWDG. In this program we will be using the WKUP pin to wake up the CPU.

Before entering this mode, the **PWR\_FLAG** and the **RTC\_FLAG** must be cleared. Since we will be using the **PWR\_FLAG**, the following HAL function must be called:

```
__HAL_PWR_CLEAR_FLAG(PWR_FLAG_WU);
```

To enable the wakeup pin 1 (GPIO pin PA0), the following HAL function should be called:

```
HAL_PWR_EnableWakeUpPin(PWR_WAKEUP_PIN1);
```

The Standby mode is entered by calling the following HAL function:

```
HAL_PWR_EnterSTANDBYMode();
```

The following statement can be used to determine whether a Reset or a wakeup from Standby mode has occurred. The statements inside the **if** block will be executed if wakeup from standby has occurred:

```
if(__HAL_PWR_GET_FLAG(PWR_FLAG_SB) != RESET)
{
}
```

Figure 15.14 shows the program listing, where only the main program is shown. At the beginning of the program we determine whether a Reset or a wakeup from Standby mode has occurred. If a wakeup occurred then the **PWR\_FLAG\_SB** is cleared, the LED is toggled, and the wakeup pin is disabled. The code outside the **if** block is executed when a Reset occurs and also after the **if** block is executed. Here, the power flag is cleared, wakeup is enabled,

and the CPU is put into Standby mode.

```
#include «main.h»

void SystemClock_Config(void);
static void MX_GPIO_Init(void);

int main(void)
{
 HAL_Init();
 SystemClock_Config();
 MX_GPIO_Init();

 if(__HAL_PWR_GET_FLAG(PWR_FLAG_SB) != RESET) // If Wakeup from Standby
 {
 __HAL_PWR_CLEAR_FLAG(PWR_FLAG_SB); // Clear PWR_FLAG_SB
 HAL_GPIO_TogglePin(GPIOA, LED_Pin);
 HAL_PWR_DisableWakeUpPin(PWR_WAKEUP_PIN1); // Disable WAKEUP_PIN1(PA0)
 }

 __HAL_PWR_CLEAR_FLAG(PWR_FLAG_WU); // Clear PWR_FLAG
 HAL_PWR_EnableWakeUpPin(PWR_WAKEUP_PIN1); // Enable WAKEUP_PIN1(PA0)
 HAL_PWR_EnterSTANDBYMode();

 while (1)
 {
 }
}
```

*Figure 15.14: Program listing.*

## 15.10 Summary

In this Chapter we have learned the various power modes of the STM32L476RG MCU. Several example projects have been developed to show how to use the sleep mode, stop mode, and the standby mode. In the next Chapter we will be focusing on the Nucleo Expansion Boards and develop projects using various expansion boards.

## CHAPTER 16 • Using the Expansion Boards

### 16.1 Overview

There are large number of Expansion Boards that can be used with the STM32 Nucleo development boards. The expansion boards plug on top of the Nucleo boards through the Arduino connectors. Some boards include sensors (e.g. magnetometer, accelerometer, temperature sensor, and humidity sensor), some include electric motor drivers, some provide Wi-Fi capabilities to the Nucleo boards, some provide industrial outputs and so on. The expansion boards are equipped with standardized interconnections including

- an Arduino Uno R3 connector, or
- a Morpho connector for a higher level of connectivity.

STM32 Nucleo expansion boards carry all the required components to evaluate various ST development boards, as well as help to speed up the project development cycle by providing tested hardware and software solutions.

In this Chapter we shall be using some of these expansion boards in simple projects. The aim here is not to develop complex projects, but to show how these boards can be used with the Nucleo-L476RG development board in projects. More information on the Nucleo expansion boards can be obtained from the following site:

<http://www.st.com/en/evaluation-tools/stm32-nucleo-expansion-boards.html?querycriteria=productId=SC1971>

All pictures of Expansion Boards in this Chapter are the copyright of STMicroelectronics and have been used here with their written permission: ©STMicroelectronics. *Used with permission.*

### 16.2 Industrial Digital Output Expansion Board (X-NUCLEO-OUT01A1)

This is an industrial output expansion board based on the ISO8200BQ chip. This board provides 8 outputs with status LED indicators. The operating voltage is 10.5 V to 33 V, and the maximum output current per channel is 700 mA. The board uses the ISO8200BQ galvanic isolated octal power solid state relay, making it possible for use in power control applications. It contains two independent galvanic isolated voltage domains with GND protection, over-voltage and under-voltage shutdown and over-temperature protection.

ISO8200BQ can drive any kind of load with one side connected to ground. Eight screw type output connectors are provided on the board for making connections to external world. Figure 16.1 shows the PCB layout of the board. The ISO8200BQ chip is placed in the middle of the board. The screw type output connector (J5) is at the bottom of the board. At the top of the board a two-terminal screw connector (J2) is provided for applying external digital voltage (7 V – 12 V) should it be necessary, for example in standalone battery operations. An external 10.5 V to 33 V power supply must be connected to the two-terminal connector J1, placed at the bottom right of the board. A jumper selection (J4) is provided next to the chip for selecting the operation mode between SYNC and DIRECT as we shall see later. For DIRECT access mode jumper J4 should be connected to GND, and for SYNC access it should be connected to +3.3 V.



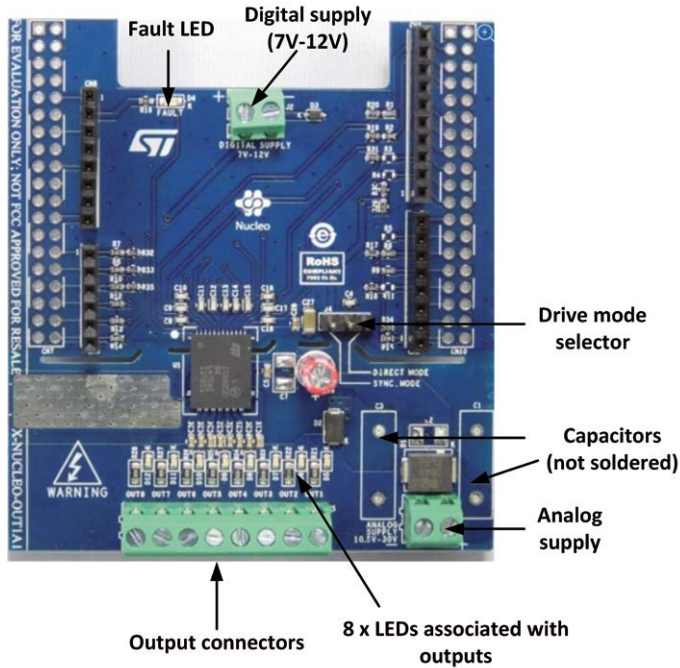


Figure 16.1: PCB Layout of the industrial digital output expansion board.]

Before using the board, it's important to understand how the ISO8200BQ solid state relay chip operates. Figure 16.2 shows the block diagram of the ISO8200BQ chip. On the left we can see eight inputs labelled IN1 to IN8, and on the right-hand side one of the eight outputs is shown. IN1 to IN8 are the driving signals for the corresponding output channels. 10.5 V to 30 V external voltage is applied to the chip through its high voltage Vdd pin. Vcc is the external low voltage logic power pin. We can also see control pins labelled SYNC, LOAD, OUT\_EN, and FAULT. The FAULT pin is an open-drain active low fault indication pin and is connected to an LED on the expansion board.

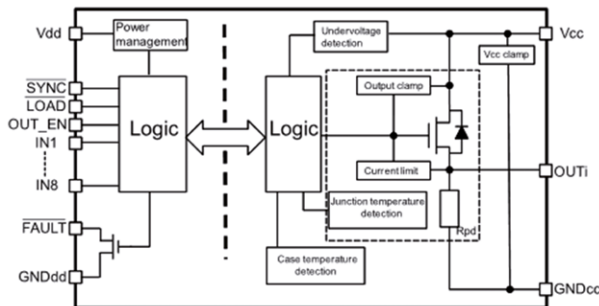


Figure 16.2: Block diagram of ISO8200BQ.

The functions of the control pins are as follows:

**LOAD:** This pin is active low, and it loads the input data into the input buffer.

**SYNC:** This pin is active low input-output synchronization pin and it copies the input buffer into the transmission buffer and manages synchronization between the low voltage side and the channel outputs on the isolated the high voltage side.

**OUT\_EN:** This is active high output enable pin which when set enables the channel outputs.

Table 16.1 shows the operation of the ISO8200BQ chip in terms of its control pins.

| LOAD | SYNC | OUT_EN | Description                                                                                      |
|------|------|--------|--------------------------------------------------------------------------------------------------|
| X    | X    | 0      | Outputs are disabled (OFF)                                                                       |
| 1    | 1    | 1      | Outputs are left unchanged                                                                       |
| 0    | 1    | 1      | Input buffer is enabled<br>Outputs are left unchanged                                            |
| 1    | 0    | 1      | Internal refresh signal disabled<br>Transmission buffer is updated<br>Outputs are left unchanged |
| 0    | 0    | 1      | The device operates in DIRECT control mode                                                       |

*Table 16.1: Operation of the ISO8200BQ.*

In DIRECT control mode each input **IN<sub>i</sub>** drives its corresponding output channel **OUT<sub>i</sub>**. The chip can also be operated in SYNCHRONOUS control mode (SCM) when the LOAD input is 0 (which updates the input buffer) and the SYNC input is 1. Then, LOAD input is set to 1 and SYNC is cleared to 0 to update the transmission buffer without changing the outputs. This mode of operation reduces the jittering of the outputs when we drive all the outputs for different devices at the same time.

Jumper J4 next to the chip on the expansion board is used to select between the DIRECT or SYNCHRONOUS modes of operation. In this Chapter we shall be using the DIRECT mode by connecting the right two positions of J4 with a connector.

The connections between the Nucleo-L476 processor GPIO pins and the expansion board pins are as follows (only the pins of interest are shown here):

| Expansion board signal | Nucleo-L476 GPIO pin |
|------------------------|----------------------|
| IN1                    | PA0                  |
| IN2                    | PA1                  |
| IN3                    | PA3                  |
| IN4                    | PA4                  |
| IN5                    | PB4                  |
| IN6                    | PB5                  |
| IN7                    | PB9                  |

|        |     |
|--------|-----|
| IN8    | PB8 |
| LOAD   | PC0 |
| SYNC   | PC1 |
| OUT_EN | PB0 |

Figure 16.3 shows the functional block diagram of the industrial digital output expansion board.

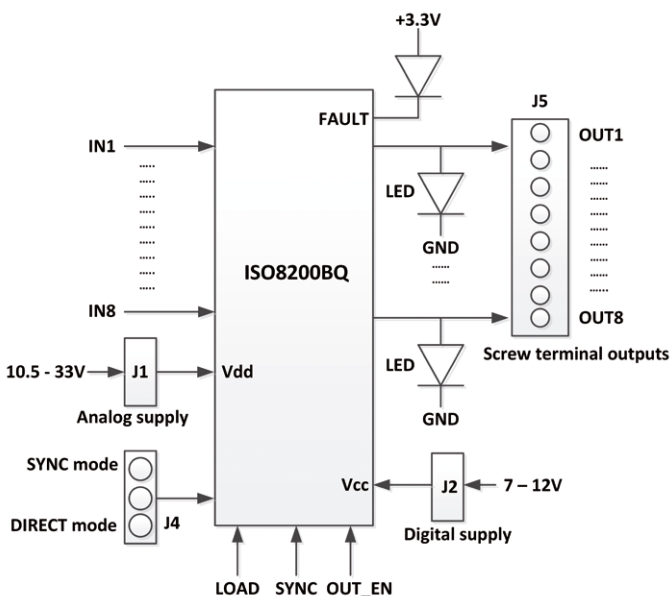


Figure 16.3: Block diagram of the industrial digital output expansion board.

More information on the industrial digital output expansion board can be obtained from the following website:

[http://www.st.com/content/st\\_com/en/products/ecosystems/stm32-open-development-environment/stm32-nucleo-expansion-boards/stm32-ode-translate-hw/x-nucleo-out01a1.html](http://www.st.com/content/st_com/en/products/ecosystems/stm32-open-development-environment/stm32-nucleo-expansion-boards/stm32-ode-translate-hw/x-nucleo-out01a1.html)

### 16.3 Project 1: Flashing an LED

#### Description

This is a simple project where we will be sending data to digital output 1 (OUT1) of the industrial digital output expansion board to flash the LED connected to this output every second. No external hardware is used in this project except a 12 V power supply. In real-time control applications a load (e.g. an actuator, a light, or a motor) can be connected to the output terminals.

#### The aim

The aim of this project is to show how the industrial digital output expansion board can be controlled using a simple project as an example.

**Block diagram**

The expansion board is simply plugged on top of the Nucleo-L476RG board.

**The hardware**

A 12 VDC external power supply must be connected to jumper J1 of the expansion board, located at the bottom right-hand side of the board.

**Program listing**

The steps are given below.

- Start the STM32CubeIDE program.
- Create a new workspace.
- Start new project.
- Select the Nucleo-L476RG board as before.
- Name the program as **INDIO**.
- Click on pins **PA0** to **PB0** as shown above and configure these pins as **GPIO\_Output**. Click right mouse and give the following labels to the pins (Figure 16.4):

| Pin | Label  |
|-----|--------|
| PA0 | IN1    |
| PA1 | IN2    |
| PA3 | IN3    |
| PA4 | IN4    |
| PB4 | IN5    |
| PB5 | IN6    |
| PB9 | IN7    |
| PB8 | IN8    |
| PC0 | LOAD   |
| PC1 | SYNC   |
| PB0 | OUT_EN |

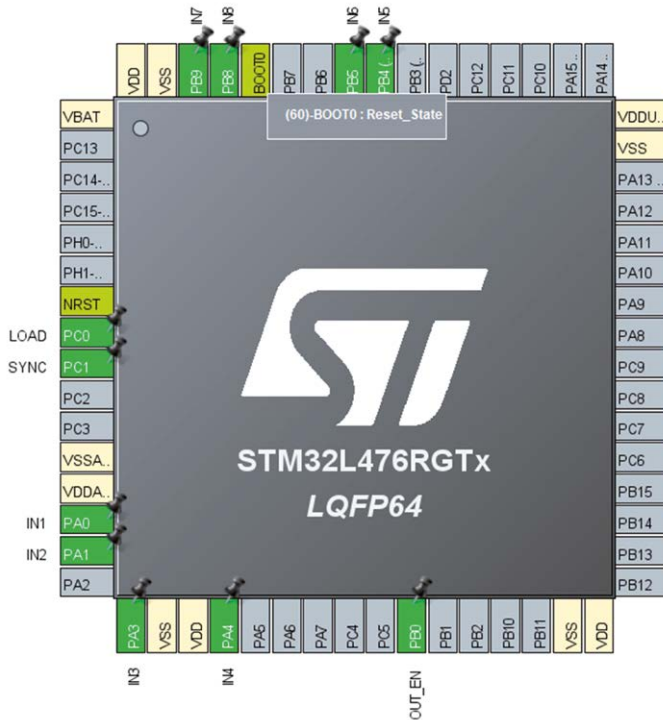


Figure 16.4: Configuring the GPIO pins.

- Click **Clock Configuration** and make sure the system clock has been configured to 80 MHz.
- Click **File**, followed by **Save** and click **YES** to generate code.
- Click **Core**, followed by **Src**, and double click **main.c** to open the main program.

Outside the main program loop the board is set to DIRECT mode by clearing pins **LOAD** and **SYNC**, and setting pin **OUT\_EN**. Inside the main program loop OUT1 is toggled every second by toggling the input pin IN1 so that the LED connected to OUT1 flashes every second. Figure 16.5 shows the program (folder: **INDIO**) listing. The clock and error routines are not shown here.

```
#include «main.h»

void SystemClock_Config(void);
static void MX_GPIO_Init(void);

int main(void)
{
 HAL_Init();
 SystemClock_Config();
```

```

MX_GPIO_Init();

//
// Configure the control pins to DIRECT mode
//
HAL_GPIO_WritePin(GPIOC, LOAD_Pin, GPIO_PIN_RESET);
HAL_GPIO_WritePin(GPIOA, SYNC_Pin, GPIO_PIN_RESET);
HAL_GPIO_WritePin(GPIOB, OUT_EN_Pin, GPIO_PIN_SET);

while (1)
{
 HAL_GPIO_WritePin(GPIOA, IN1_Pin, GPIO_PIN_RESET); // LED OFF
 HAL_Delay(1000); // 1 sec delay
 HAL_GPIO_WritePin(GPIOA, IN1_Pin, GPIO_PIN_SET); // LED ON
 HAL_Delay(1000); // 1 sec delay
}
}

```

*Figure 16.5: Program listing.*

#### 16.4 Brushed DC Motor Driver Expansion Board (X-NUCLEO-IHM13A1)

This expansion board (see Figure 16.6) is based on STSPIN250 driver chip and can be used in small, brushed DC motor control applications. The features of the board are:

- compatible with Nucleo boards;
- low voltage range from 1.8 V to 10 V;
- current up to 2.6 A<sub>rms</sub>;
- current control with adjustable off-time;
- full overcurrent and short circuit protection;
- thermal shutdown;
- compatible with Arduino UNO R3 connector.



*Figure 16.6: Brushed DC Motor Driver Expansion Board.*

Figure 16.7 shows the component layout on the expansion board. The board is connected to the Nucleo boards through the Arduino connectors CN5, CN9, CN6 and CN8. The ST-SPIN250 driver chip is located at the centre of the board. Power to the motor is applied through the 2-terminal connector J1 (10 V maximum). The motor pins are connected to the 2-terminal connector J2. Two yellow LEDs on the board (D1 and D2) indicate the output operation. A red fault LED (D3) is also provided.

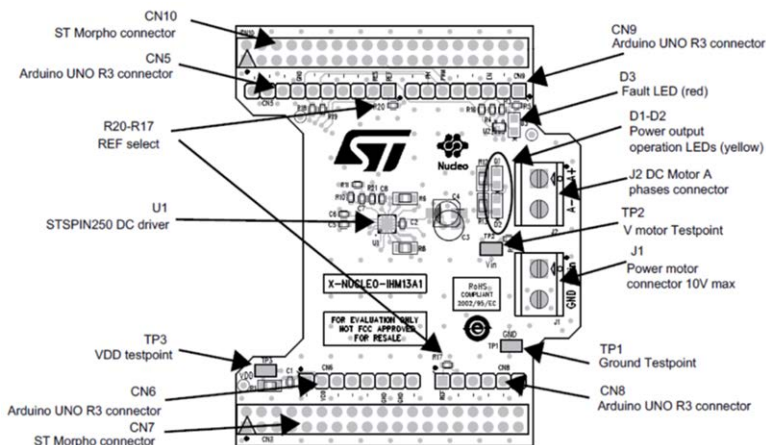


Figure 16.7: The component layout.

The following pins are used on the Arduino connectors:

| Arduino Connector | Pin no | Signal        | Nucleo-L476RG pin |
|-------------------|--------|---------------|-------------------|
| CN5               | 2      | RST (or STBY) | PC_7              |
| CN9               | 3      | EN            | PA_10             |
| CN9               | 6      | PWM           | PB_4              |
| CN9               | 7      | PH            | PB_10             |
| CN8               | 1      | REF           | PA_0              |

The following pins are the ground pins: pin 7 CN5, pin 6 and pin 7 CN6.  $V_{DD}$  is available at pin 2 of CN6.

- PWM:** This is the Pulse Width Modulated input.
- PH:** This is the logic input.
- REF:** This is the reference voltage for the current limiter circuitry.
- RST:** When forced low the device is put into low consumption mode.
- EN:** This is the power stage enable input.

When  $EN = 0$  the driver is disabled. During normal operation the following configuration settings are required:

$$EN = 1, PWM = 1$$

PH can be either 0 or 1 and it determines the direction of rotation.

Figure 16.8 shows the functional block diagram of the expansion board.

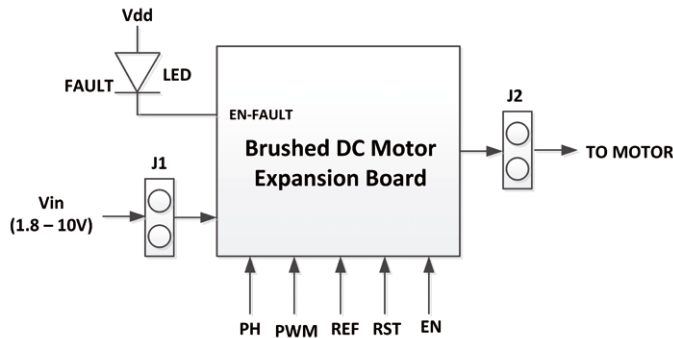


Figure 16.8: Functional block diagram of the expansion board.

Further information on this expansion board can be obtained from the following website:

[http://www.st.com/content/st\\_com/en/products/ecosystems/stm32-open-development-environment/stm32-nucleo-expansion-boards/stm32-ode-move-actuate-hw/x-nucleo-ihm13a1.html](http://www.st.com/content/st_com/en/products/ecosystems/stm32-open-development-environment/stm32-nucleo-expansion-boards/stm32-ode-move-actuate-hw/x-nucleo-ihm13a1.html)

A firmware called X-CUBE-SPN13 is available for the X-NUCLEO-1HM13A1 expansion board at the following web site (go to the end of the page and click **Get Software** to download the software):

[http://www.st.com/content/st\\_com/en/products/embedded-software/mcus-embedded-software/stm32-embedded-software/stm32cube-expansion-packages/x-cube-spn13.html](http://www.st.com/content/st_com/en/products/embedded-software/mcus-embedded-software/stm32-embedded-software/stm32cube-expansion-packages/x-cube-spn13.html)

### 16.5 Motion MEMS and Environmental Sensor Expansion Board (X-NUCLEO-IK-S01A2)

This expansion board (see Figure 16.9) communicates with the Nucleo boards through the I<sup>2</sup>C interface and it is designed around the LSM6DSL 3D accelerometer and 3D gyroscope, LSM303AGR 3D accelerometer and 3D magnetometer, the HTS221 humidity and temperature sensor and the LPS22HB pressure sensor. A DIL-24 socket is provided on the board for additional sensors.

The basic features of this board are:

- LSM6DSL MEMS 3D accelerometer ( $\pm 2/4/8/16$  g) and 3D gyroscope ( $\pm 125/245/500/1000/2000$  dps);
- LSM303AGR MEMS 3D accelerometer ( $\pm 2/4/8/16$  g) and MEMS3D magnetometer ( $\pm 50$  gauss);
- LPS22HB MEMS pressure sensor, 260-1260 hPa absolute digital output barometer;
- HTS221: capacitive digital relative humidity and temperature;
- DIL24 socket for additional MEMS adapters and other sensors;
- equipped with Arduino UNO R3 connector.



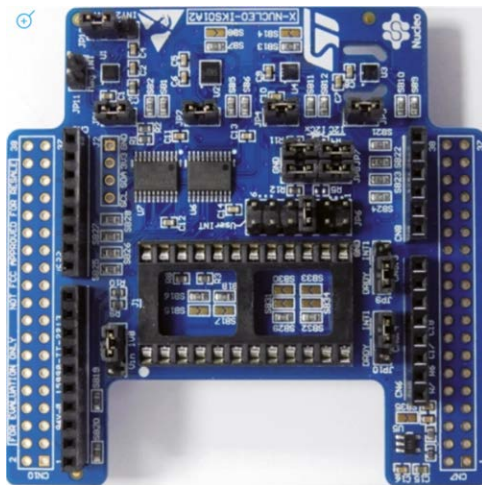


Figure 16.9: Motion MEMS and environmental sensor expansion board.

Further details about this expansion board can be obtained from the following website:  
<https://www.st.com/en/ecosystems/x-nucleo-iks01a2.html>

Figure 16.10 shows the functional block diagram of this expansion board.

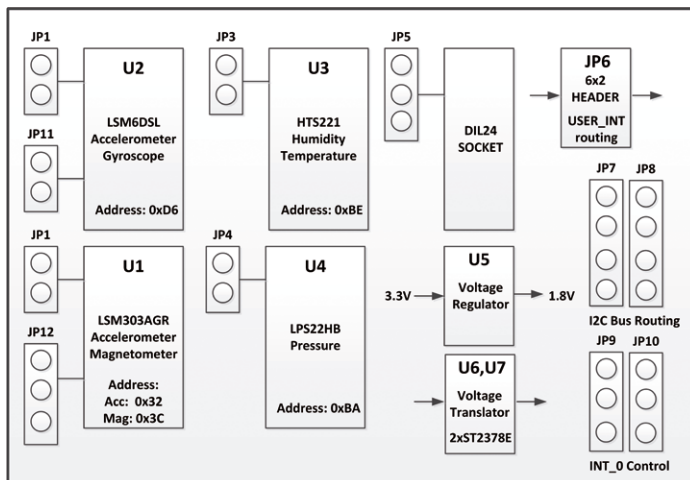


Figure 16.10: Functional block diagram.

The I<sup>2</sup>C addresses of the various sensors on the board are as follows:

|           |                           |      |
|-----------|---------------------------|------|
| LSM6DSL   | Accelerometer + Gyroscope | 0xD6 |
| LSM303AGR | Accelerometer             | 0x32 |
| LSM303AGR | Magnetometer              | 0x3C |
| HTS221    | Humidity + Temperature    | 0xBE |
| LPS22HB   | Pressure + Temperature    | 0xBA |

The board can be configured in several different I<sup>2</sup>C modes, such as connecting all sensors to the same I<sup>2</sup>C bus, making the LSM6DSL the master and the other slaves, and so on. In this book the default mode is selected where all the sensors are driven from the same I<sup>2</sup>C bus and they are all slaves of the host processor (see Figure 16.11). For this mode, all the pins of JP7:1-2-3-4 and JP8:1-2-3-4 are connected together on the board.

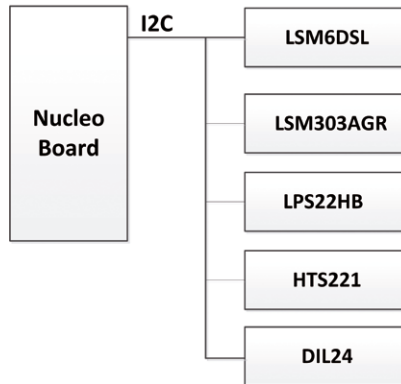


Figure 16.11: All sensors are slaves.

The current consumption of each sensor can be measured by connecting an ammeter between the following jumper pins:

|           |     |
|-----------|-----|
| LSM303AGR | JP1 |
| LSM6DSL   | JP2 |
| HTS221    | JP3 |
| LPS22HB   | JP4 |
| DIL24     | JP5 |

The following Nucleo board pins are used for the I<sup>2</sup>C connection:

| I <sup>2</sup> C pin | GPIO pin |
|----------------------|----------|
| SCL                  | PB8      |
| SDA                  | PB9      |

Interrupts can be assigned between an external board and the sensors on the expansion board. JP6 is a 6×2 jumper block that can be used to select interrupt signals where one INT signal can be connected to USER\_INT through jumper JP6 (see the expansion board User Manual for further details). The default connection is USER\_INT to HTS221 data ready signal (DRDY).

There are several example application programs and a GUI demo program available for this expansion board. Interested readers should refer to the following website:

[http://www.st.com/content/st\\_com/en/products/ecosystems/stm32-open-development-environment/stm32-nucleo-expansion-boards/stm32-ode-sense-hw/x-nucleo-iks01a2.html#sw-tools-scroll](http://www.st.com/content/st_com/en/products/ecosystems/stm32-open-development-environment/stm32-nucleo-expansion-boards/stm32-ode-sense-hw/x-nucleo-iks01a2.html#sw-tools-scroll)

## 16.6 Project 2: Reading the Temperature from the X-NUCLEO-IKS01A2 Expansion Board

### Description

In this section we will be developing a project to read the ambient temperature from the LPS22HB digital pressure/temperature sensor chip on the X-NUCLEO-IKS01A2 expansion board, and then display it on the PC screen every second. But before doing that, we have to know the operation and internal structure of the LPS22HB chip and also be familiar with programming its registers. This is described briefly in the next sections.

### The aim

The aim of this project is to show how the LPS22HB chip on the X-NUCLEO-IKS01A2 expansion board can be used to read the temperature and send it to the PC.

### The LPS22HB pressure/temperature sensor

The LPS22HB datasheet is available at the following STMicroelectronics website:  
<https://www.st.com/en/mems-and-sensors/lps22hh.html>

Brief details of this sensor are given in this section which are necessary to know before the device can be programmed.

The LPS22HB is an absolute pressure/temperature sensor which communicates through the I<sup>2</sup>C, MIPI I3CSM, or SPI interfaces. On the X-NUCLEO-IKS01A2 expansion board the LPS22HB is connected to the I<sup>2</sup>C bus and is operated in I<sup>2</sup>C mode using the SCL and SDA pins, which correspond to the I/O pins PB8 and PB9 of the microcontroller respectively. In addition to pressure output, the chip provides 16-bit temperature output, where the output data rate (ODR) can be set in software between 1 Hz and 200 Hz.

A 128-slot 40-bit FIFO is available on the device to store the pressure and temperature output values. This feature allows power saving since the host processor does not have to continuously poll data from the sensor, but it can wake up only when needed to get the data from the FIFO. In this project the FIFO is disabled by programming the correct register.

The temperature data is stored in three registers: **TEMP\_OUT\_H** (address: 0x2C), and **TEMP\_OUT\_L** (address: 0x2B). The temperature output data is in 2's complement form and the actual physical temperature reading is obtained by combining the high and low bytes into a 16-bit word and then dividing the result by 100.

An example is given here. Assume that the two registers contain the following binary data:

```
TEMP_OUT_H: 00010110 (0x16)
TEMP_OUT_L: 0000003 (0x03)
```

The temperature value in hexadecimal is: 0x1603 which is equal to 5635 as a signed decimal number. Dividing by 100 gives  $5635/100 = 56.35$  °C.

The I<sup>2</sup>C write and read addresses of the LPS22HB are set to 0xBA and 0xBB respectively. In the remaining parts of this section some tables are given which will be required for the development of the program. The register map of the LPS22HB device is shown in Table 16.1.

| Name          | Type | Register Address | Default  | Function and comment         |
|---------------|------|------------------|----------|------------------------------|
|               |      | Hex              | Binary   |                              |
| Reserved      |      | 00 - 0A          | -        | Reserved                     |
| INTERRUPT_CFG | R/W  | 0B               | 00000000 | Interrupt register           |
| THS_P_L       | R/W  | 0C               | 00000000 | Pressure threshold registers |
| THS_P_H       | R/W  | 0D               | 00000000 |                              |
| Reserved      |      | 0E               | -        | Reserved                     |
| WHO_AM_I      | R    | 0F               | 10110001 | Who am I                     |
| CTRL_REG1     | R/W  | 10               | 00000000 | Control registers            |
| CTRL_REG2     | R/W  | 11               | 00010000 |                              |
| CTRL_REG3     | R/W  | 12               | 00000000 |                              |
| Reserved      |      | 13               | -        | Reserved                     |
| FIFO_CTRL     | R/W  | 14               | 00000000 | FIFO configuration register  |
| REF_P_XL      | R/W  | 15               | 00000000 | Reference pressure registers |
| REF_P_L       | R/W  | 16               | 00000000 |                              |
| REF_P_H       | R/W  | 17               | 00000000 |                              |
| RPDS_L        | R/W  | 18               | 00000000 | Pressure offset registers    |
| RPDS_H        | R/W  | 19               | 00000000 |                              |
| RES_CONF      | R/W  | 1A               | 00000000 | Resolution register          |
| Reserved      |      | 1B - 24          | -        | Reserved                     |

|              |   |         |        |                              |
|--------------|---|---------|--------|------------------------------|
| INT_SOURCE   | R | 25      | output | Interrupt register           |
| FIFO_STATUS  | R | 26      | output | FIFO status register         |
| STATUS       | R | 27      | output | Status register              |
| PRESS_OUT_XL | R | 28      | output | Pressure output registers    |
| PRESS_OUT_L  | R | 29      | output |                              |
| PRESS_OUT_H  | R | 2A      | output |                              |
| TEMP_OUT_L   | R | 2B      | output | Temperature output registers |
| TEMP_OUT_H   | R | 2C      | output |                              |
| Reserved     |   | 2D - 32 | -      | Reserved                     |
| LPFP_RES     | R | 33      | output | Filter reset register        |

Table 16.1: LPS22HB register map, ©STMicroelectronics. Used with permission.

Some of the important register bits are described below. Interested readers can refer to page 32 of the device datasheet for detailed information on all registers of the LPS22HB.

### CTRL\_REG1 (0x10)

As shown in Table 16.2, this register is used to set the output data rate, to enable low-pass filter at the output, update mode, and the SPI serial interface mode.

| 7 | 6    | 5    | 4    | 3       | 2        | 1   | 0   |
|---|------|------|------|---------|----------|-----|-----|
| 0 | ODR2 | ODR1 | ODR0 | EN_LPFP | LPFP_CFG | BDU | SIM |

|                    |                                                                                                                                                        |
|--------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------|
| ODR[2:0]           | Output data rate selection. Default value: 000<br>Refer to <a href="#">Table 18</a> .                                                                  |
| EN_LPFP            | Enable low-pass filter on pressure data when Continuous mode is used.<br>Default value: 0<br>(0: Low-pass filter disabled; 1: Low-pass filter enabled) |
| LPFP_CFG           | LPFP_CFG: Low-pass configuration register. Default value: 0<br>Refer to <a href="#">Table 19</a> .                                                     |
| BDU <sup>(1)</sup> | Block data update. Default value: 0<br>(0: continuous update;<br>1: output registers not updated until MSB and LSB have been read)                     |
| SIM                | SPI Serial Interface Mode selection. Default value: 0<br>(0: 4-wire interface; 1: 3-wire interface)                                                    |

Table 16.2: CTRL\_REG1, ©STMicroelectronics. Used with permission.

The output data rate is selected as shown in Table 16.3.

| ODR[2:0]           | Temperature, Pressure |
|--------------------|-----------------------|
| 000                | One-shot              |
| 001                | 1 Hz                  |
| 010                | 10 Hz                 |
| 011                | 25 Hz                 |
| 100                | 50 Hz                 |
| 101                | 75 Hz                 |
| 110 <sup>(1)</sup> | 100 Hz                |
| 111 <sup>(1)</sup> | 200 Hz                |

Table 16.3: Selecting the output data rate, ©STMicroelectronics. Used with permission.

### FIFO\_CTRL (0x13)

This is the FIFO control register whose bits are given in Table 16.4.

|         |         |         |      |      |      |      |      |
|---------|---------|---------|------|------|------|------|------|
| 7       | 6       | 5       | 4    | 3    | 2    | 1    | 0    |
| F_MODE2 | F_MODE1 | F_MODE0 | WTM4 | WTM3 | WTM2 | WTM1 | WTM0 |

|             |                                                                                                                                    |
|-------------|------------------------------------------------------------------------------------------------------------------------------------|
| F_MODE[2:0] | FIFO mode selection. Default value: 000<br>Refer to <a href="#">Table 20</a> and <a href="#">Section 5</a> for additional details. |
| WTM[4:0]    | FIFO watermark level selection.                                                                                                    |

Table 16.4: FIFO\_CTRL register bits, ©STMicroelectronics. Used with permission.

The FIFO mode is selected as shown in Table 16.5.

| F_MODE2 | F_MODE1 | F_MODE0 | FIFO mode selection   |
|---------|---------|---------|-----------------------|
| 0       | 0       | 0       | Bypass mode           |
| 0       | 0       | 1       | FIFO mode             |
| 0       | 1       | 0       | Stream mode           |
| 0       | 1       | 1       | Stream-to-FIFO mode   |
| 1       | 0       | 0       | Bypass-to-Stream mode |
| 1       | 0       | 1       | Reserved              |
| 1       | 1       | 0       | Dynamic-Stream mode   |
| 1       | 1       | 1       | Bypass-to-FIFO mode   |

Table 16.5 Selecting the FIFO mode, © STMicroelectronics. Used with permission.

## STATUS (0x27)

This read-only register is used to find out if new pressure and temperature data are available, as well as if there has been data overrun. Table 6.6 shows the bits of this register.

|    |    |      |      |    |    |      |      |
|----|----|------|------|----|----|------|------|
| 7  | 6  | 5    | 4    | 3  | 2  | 1    | 0    |
| -- | -- | T_OR | P_OR | -- | -- | T_DA | P_DA |

|      |                                                                                                                               |
|------|-------------------------------------------------------------------------------------------------------------------------------|
| T_OR | Temperature data overrun.<br>(0: no overrun has occurred;<br>1: a new data for temperature has overwritten the previous data) |
| P_OR | Pressure data overrun.<br>(0: no overrun has occurred;<br>1: new data for pressure has overwritten the previous data)         |
| T_DA | Temperature data available.<br>(0: new data for temperature is not yet available;<br>1: a new temperature data is generated)  |
| P_DA | Pressure data available.<br>(0: new data for pressure is not yet available;<br>1: a new pressure data is generated)           |

Table 6.6 STATUS register bits, ©STMicroelectronics. Used with permission.

## Program listing

A firmware called X-CUBE-MEMS1 is available for the X-NUCLEO-IKS01A2 expansion board at the following website (go to the end of the page and click **Get Software** to download the software). At the time of writing this book the X-CUBE-MEMS1 was called: **en.x-cube-mems1.zip**. You should unzip the file in a folder.

[http://www.st.com/content/st\\_com/en/products/embedded-software/mcus-embedded-software/stm32-embedded-software/stm32cube-expansion-packages/x-cube-mems1.html](http://www.st.com/content/st_com/en/products/embedded-software/mcus-embedded-software/stm32-embedded-software/stm32cube-expansion-packages/x-cube-mems1.html)

The firmware contains all the project codes for an application, developed for different types of Nucleo boards. In this example project we will be using the Nucleo-L476RG development board. There are example projects using the various sensors on the expansion board.

First, we will have to install the X-CUBE-MEMS1 library. The steps are given below.

- Start the STM32Cube IDE.
- Start a new STM32 project.
- Create a new workspace.
- Select STM32L476RG as the processor.
- Name the program as **MEM**.
- Click **Help**, followed by **Manage embedded software packages**.
- Press **Refresh** button to get an updated list of add-on packs.
- Click **STMicroelectronics** tab and Locate **X-CUBE-MEMS1** pack (Figure 16.12).

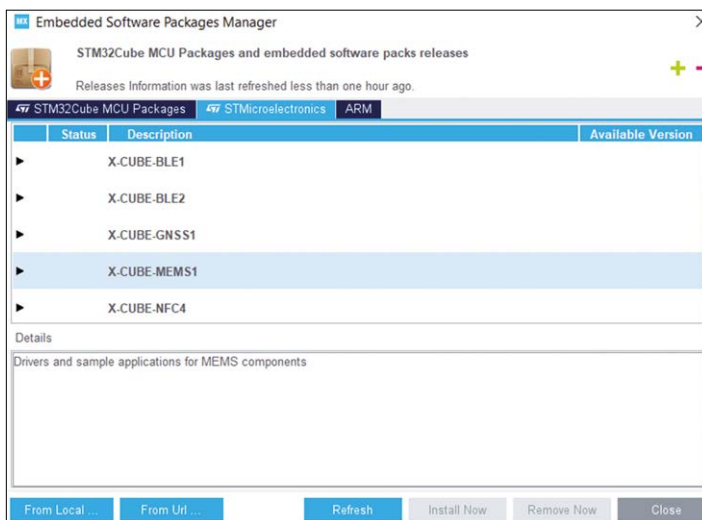


Figure 16.12: Locate X-CUBE-MEMS1.

- Click to expand and then Select **Drivers and sample applications for MEMS components**.
- Click to install.

- To add the X-CUBE-MEMS1 software pack to the project, click the **Software Packs** button and then the **Select Components** item.
- Select **LPS22HB** and **I2C** on the right-hand side (Figure 16.13).

| Pack / Bundle / Component      | Version | Selection    |
|--------------------------------|---------|--------------|
| Board Part AccGyr / LSM6DSOX   |         | Not selected |
| Board Part AccGyr / LSM6DSO32  |         | Not selected |
| Board Part AccMag / LSM303AGR  |         | Not selected |
| Board Part AccMag / LSM303DAC  |         | Not selected |
| Board Part Acc / LIS2DW12      |         | Not selected |
| Board Part Acc / LIS2DH12      |         | Not selected |
| Board Part Acc / IIS2DLPC      |         | Not selected |
| Board Part Acc / AIS2DW12      |         | Not selected |
| Board Part Acc / AIS328DQ      |         | Not selected |
| Board Part Acc / AIS3624DQ     |         | Not selected |
| Board Part Acc / H3LIS331DL    |         | Not selected |
| Board Part Acc / IIS2ICLX      |         | Not selected |
| Board Part Mag / LIS3MDL       |         | Not selected |
| Board Part Mag / LIS2MDL       |         | Not selected |
| Board Part Mag / IIS2MDC       |         | Not selected |
| Board Part HumTemp / HTS221    |         | Not selected |
| Board Part PressTemp / LPS22HB | 5.2.2   | I2C          |
| Board Part PressTemp / LPS22HH |         | Not selected |
| Board Part PressTemp / LPS33HW |         | Not selected |
| Board Part PressTemp / LPS33K  |         | Not selected |

Figure 16.13: Select LPS22HB with I<sup>2</sup>C.

We can now start writing our program. In this section, the program has been developed from the first principles using registers of the LPS22HB sensor.

The steps are as follows.

- Configure the I<sup>2</sup>C bus. Enter **PB8** at the bottom right-hand side search for this pin in the pin layout. Click on the flashing pin and select **I2C1\_SCL**.
- Now find pin **PB9** and set it to **I2C1\_SDA**.
- At this point the I2C pins are selected but not configured yet. Expand the **Connectivity** tab at the left-hand side and enable **I2C1**. The two pins on the pin layout should turn green.
- Set the system clock to 80 MHz.
- Select **PA2** and configure it as **USART2\_TX**. Also, select **PA3** and configure it as **USART2\_RX** (not used in this project).
- Expand the **Connectivity** tab and select **USART2**. Set its **Mode** to **Asynchronous**.
- In **Parameter Settings**, set the **Baud Rate** to 9600, **Word Length** to 8, **Parity** to None, and **Stop Bits** to 1 (Figure 16.14).



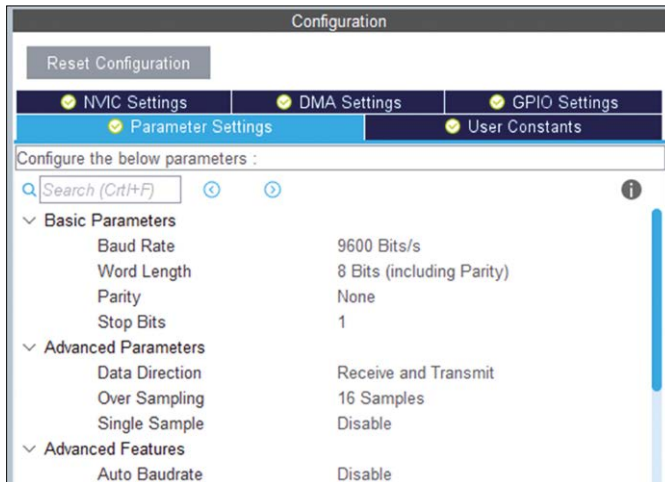


Figure 16.14: Configuring USART2.

- Click **File**, followed by **Save** and click **YES** to generate code.
- Click **Core**, followed by **Src** and double click **main.c** to open the main program.

Type the sensor register addresses to the beginning of the program. Only the registers used in the program are specified here:

```
#define LPS22HBW 0xBA
#define LPS22HBR 0xBB
#define CTRL_REG1 0x10
#define CTRL_REG2 0x11
#define FIFO_CTRL 0x13
#define STATUS 0x27
```

Also, insert the following statements to the beginning of the program:

```
int tempC;
uint8_t Dat[2];
```

Create a function called **LPS22HB\_GetStatus** to return the status of the sensor by accessing the register **STATUS** at address 0x27. Notice that function **HAL\_I2C\_Master\_Transmit** is used to send data to the sensor over the I<sup>2</sup>C bus. Writing data to an I<sup>2</sup>C sensor requires only one step where the required register is addressed, and the data is sent. Reading data from an I<sup>2</sup>C sensor is in two steps where firstly we send the address in write mode, and then read the data in read mode. The **HAL\_I2C\_Master\_Transmit** has the parameters: I<sup>2</sup>C handle, device address, register address, number of bytes, timeout. Similarly, **HAL\_I2C\_Master\_Receive** has the parameters: I<sup>2</sup>C handle, device address, data buffer to received returned data, number of bytes to be returned, timeout:

```

uint8_t LPS22HB_GetStatus(void)
{
 uint8_t byte;
 byte=0x27;
 HAL_I2C_Master_Transmit(&hi2c1, LPS22HHW, &byte, 1, 500);
 HAL_I2C_Master_Receive(&hi2c1, LPS22HHR, Dat,1, 500);
 return Dat[0];
}

```

Create a function called **LPS22HB\_Init** to initialize the sensor registers. As shown below, the registers are initialized as follows:

| Register  | Data | Action                                                  |
|-----------|------|---------------------------------------------------------|
| FIFO_CTRL | 0x02 | Disable FIFO                                            |
| CTRL_REG1 | 0x3A | ODR to 25 Hz, enable LPF, update output after data read |
| CTRL_REG2 | 0x10 | Increment register address automatically                |

Data is read from the sensor using function **LPS22HB\_GetTemperature**. Notice that the program waits until data is ready (bit 2 of STATUS register is set to 1):

```

int LPS22HB_GetTemperature(void)
{
 uint16_t temperature;
 int temp;

 while ((LPS22HH_GetStatus() & 2) == 0)
 HAL_Delay(100);

 Dat[0] = 0x2B;
 HAL_I2C_Master_Transmit(&hi2c1, LPS22HHW, Dat, 1, 500);
 HAL_I2C_Master_Receive(&hi2c1, LPS22HHR, Dat,2,500);
 temperature = (((uint16_t)Dat[1]) << 8) | (uint16_t) Dat[0];
 temp = temperature / 100.0;
 return temp;
}

```

The code inside the program loop sends a heading to the PC, gets the temperature reading as a decimal number, sends the reading to the PC with a newline. This process is repeated every second.

Figure 16.15 shows example temperature readings displayed on the terminal emulator screen of the PC. In this project the HyperTrm terminal emulator software was used.

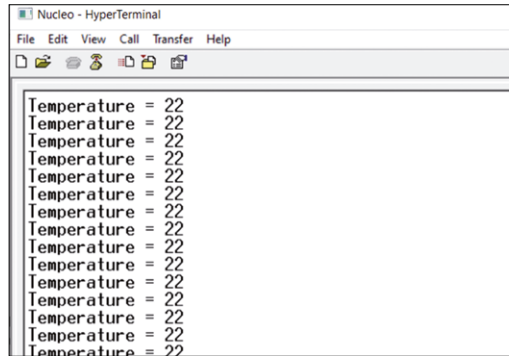


Figure 16.15: Example display on the screen.

The program listing (Program: MEM) is shown in Figure 16.16, where the comment lines have been removed for clarity. Figure 16.17 shows the X-NUCLEO-IKS01A2 expansion board plugged on top of the Nucleo\_L476RG development board.

```

/* USER CODE BEGIN Header */
/**

 * @file : main.c
 * @brief : Main program body

 * @attention
 *
 * <h2><center>© Copyright (c) 2020 STMicroelectronics.
 * All rights reserved.</center></h2>
 *
 * This software component is licensed by ST under BSD 3-Clause license,
 * the «License»; You may not use this file except in compliance with the
 * License. You may obtain a copy of the License at:
 *
 * opensource.org/licenses/BSD-3-Clause
 *

 */
#include «main.h»
#include «stdio.h»
#include «string.h»
#include «stdlib.h»

//
// Define LPS22HW register addresses
//
#define LPS22HBW 0xBA
#define LPS22HBR 0xBB
#define CTRL_REG1 0x10

```

```
#define CTRL_REG2 0x11
#define FIFO_CTRL 0x13
#define STATUS 0x27

I2C_HandleTypeDef hi2c1;
UART_HandleTypeDef huart2;

void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_I2C1_Init(void);
static void MX_USART2_UART_Init(void);

int tempC;
uint8_t Dat[2];

//
// This function returns the LPS22HB status
//
uint8_t LPS22HB_GetStatus(void)
{
 uint8_t byte;
 byte=0x27;
 HAL_I2C_Master_Transmit(&hi2c1, LPS22HBW, &byte, 1, 500);
 HAL_I2C_Master_Receive(&hi2c1, LPS22HBR, Dat,1, 500);
 return Dat[0];
}

//
// This function initializes the LPS22HB
//
void LPS22HB_Init(void)
{
 Dat[0] = CTRL_REG1;
 Dat[1] = 0x3A;
 HAL_I2C_Master_Transmit(&hi2c1, LPS22HBW, Dat, 2, 500);

 Dat[0] = CTRL_REG2;
 Dat[1] = 0x10;
 HAL_I2C_Master_Transmit(&hi2c1, LPS22HBW, Dat, 2, 500);

 Dat[0] = FIFO_CTRL;
 Dat[1] = 0x02;
 HAL_I2C_Master_Transmit(&hi2c1, LPS22HBW, Dat, 2, 500);
}

//
```

```
// This function returns the temperature
//
int LPS22HB_GetTemperature(void)
{
 uint16_t temperature;
 int temp;

 while ((LPS22HB_GetStatus() & 2) == 0)
 HAL_Delay(100);

 Dat[0] = 0x2B;
 HAL_I2C_Master_Transmit(&hi2c1, LPS22HBW, Dat, 1, 500);
 HAL_I2C_Master_Receive(&hi2c1, LPS22HBR, Dat, 2, 500);
 temperature = (((uint16_t)Dat[1]) << 8) | (uint16_t) Dat[0];
 temp = temperature / 100.0;
 return temp;
}

//
// Send a newline to PC
//
void UART_SEND_NL(UART_HandleTypeDef *huart)
{
 HAL_UART_Transmit(huart, (uint8_t*)»\n\r«, 2, HAL_MAX_DELAY);
}

//
// Send an integer number to PC
//
void UART_SEND_INT(UART_HandleTypeDef *huart, int i, int m)
{
 char buffer[10];
 itoa(i, buffer, 10);
 HAL_UART_Transmit(huart, (uint8_t*) buffer, strlen(buffer), HAL_MAX_DELAY);
 if(m == 1) HAL_UART_Transmit(huart, (uint8_t*)»\n\r«, 2, HAL_MAX_DELAY);
}

//
// Send text to PC
//
void UART_SEND_TXT(UART_HandleTypeDef *huart, char buffer[], int m)
{
 HAL_UART_Transmit(huart, (uint8_t*) buffer, strlen(buffer), HAL_MAX_DELAY);
 if(m == 1) HAL_UART_Transmit(huart, (uint8_t*)»\n\r«, 2, HAL_MAX_DELAY);
}
```

```

//
// Start of main program
//
int main(void)
{
 HAL_Init();
 SystemClock_Config();
 MX_GPIO_Init();
 MX_I2C1_Init();
 HAL_I2C_Init(&hi2c1);
 MX_USART2_UART_Init();

 LPS22HB_Init();

 while (1)
 {
 UART_SEND_TXT(&huart2, «Temperature = », 0); // Send heading
 tempC = LPS22HB_GetTemperature(); // GEt temp
 UART_SEND_INT(&huart2, tempC, 1); // Send temp
 HAL_Delay(1000); // Wait 1 sec
 }
}

void SystemClock_Config(void)
{
 RCC_OscInitTypeDef RCC_OscInitStruct = {0};
 RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};
 RCC_PeriphCLKInitTypeDef PeriphClkInit = {0};

 RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
 RCC_OscInitStruct.HSISState = RCC_HSI_ON;
 RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
 RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
 RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
 RCC_OscInitStruct.PLL.PLLM = 2;
 RCC_OscInitStruct.PLL.PLLN = 20;
 RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV7;
 RCC_OscInitStruct.PLL.PLLQ = RCC_PLLQ_DIV2;
 RCC_OscInitStruct.PLL.PLLR = RCC_PLLR_DIV2;
 if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
 {
 Error_Handler();
 }

 RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
 |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;

```

```
RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_4) != HAL_OK)
{
 Error_Handler();
}
PeriphClkInit.PeriphClockSelection = RCC_PERIPHCLK_USART2|RCC_PERIPHCLK_I2C1;
PeriphClkInit.Usart2ClockSelection = RCC_USART2CLKSOURCE_PCLK1;
PeriphClkInit.I2c1ClockSelection = RCC_I2C1CLKSOURCE_PCLK1;
if (HAL_RCCEx_PeriphCLKConfig(&PeriphClkInit) != HAL_OK)
{
 Error_Handler();
}

if (HAL_PWREx_ControlVoltageScaling(PWR_REGULATOR_VOLTAGE_SCALE1) != HAL_OK)
{
 Error_Handler();
}
}

static void MX_I2C1_Init(void)
{
 hi2c1.Instance = I2C1;
 hi2c1.Init.Timing = 0x10909CEC;
 hi2c1.Init.OwnAddress1 = 0;
 hi2c1.Init.AddressingMode = I2C_ADDRESSINGMODE_7BIT;
 hi2c1.Init.DualAddressMode = I2C_DUALADDRESS_DISABLE;
 hi2c1.Init.OwnAddress2 = 0;
 hi2c1.Init.OwnAddress2Masks = I2C_OA2_NOMASK;
 hi2c1.Init.GeneralCallMode = I2C_GENERALCALL_DISABLE;
 hi2c1.Init.NoStretchMode = I2C_NOSTRETCH_DISABLE;
 if (HAL_I2C_Init(&hi2c1) != HAL_OK)
 {
 Error_Handler();
 }

 if (HAL_I2CEx_ConfigAnalogFilter(&hi2c1, I2C_ANALOGFILTER_ENABLE) != HAL_OK)
 {
 Error_Handler();
 }

 if (HAL_I2CEx_ConfigDigitalFilter(&hi2c1, 0) != HAL_OK)
 {

```

```

 Error_Handler();
}
}

static void MX_USART2_UART_Init(void)
{
 huart2.Instance = USART2;
 huart2.Init.BaudRate = 9600;
 huart2.Init.WordLength = UART_WORDLENGTH_8B;
 huart2.Init.StopBits = UART_STOPBITS_1;
 huart2.Init.Parity = UART_PARITY_NONE;
 huart2.Init.Mode = UART_MODE_TX_RX;
 huart2.Init.HwFlowCtl = UART_HWCONTROL_NONE;
 huart2.Init.OverSampling = UART_OVERSAMPLING_16;
 huart2.Init.OneBitSampling = UART_ONE_BIT_SAMPLE_DISABLE;
 huart2.AdvancedInit.AdvFeatureInit = UART_ADVFEATURE_NO_INIT;
 if (HAL_UART_Init(&huart2) != HAL_OK)
 {
 Error_Handler();
 }
}

static void MX_GPIO_Init(void)
{
 /* GPIO Ports Clock Enable */
 __HAL_RCC_GPIOA_CLK_ENABLE();
 __HAL_RCC_GPIOB_CLK_ENABLE();
}

void Error_Handler(void)
{
}

#ifdef USE_FULL_ASSERT

void assert_failed(uint8_t *file, uint32_t line)
{
}

#endif

/***** (C) COPYRIGHT STMicroelectronics *****END OF FILE*****/

```

Figure 16.16: Program: MEM.



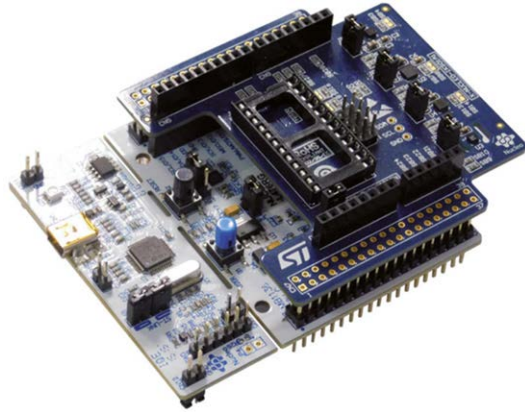


Figure 16.17: X-NUCLEO-IKS01A2 board and development board.

### 16.7 Project 3: Using the X-CUBE-MEMS1 Library

In the previous project we have developed the program from the first principles by accessing the device registers directly. In this project we will use the X-CUBE-MEMS1 library to read sensors of the X-NUCLEO-IKS01A2 expansion board and then display the data on the PC.

This project assumes that you have already installed the X-CUBE-MEMS1 library to your STM32CubeIDE as described in the previous project.

The steps are as follows.

- Start the STM32CubeIDE and create a new workspace.
- Start a new STM32 project.
- Select STM32L476RG as the processor.
- Name the program as **MEMS2**.
- Configure pin **PB8** as **I2C1\_SCL** and pin **PB9** as **I2C1\_SDA** as described in the previous project.
- Expand the **Connectivity** tab at the left-hand side and enable **I2C1**. The two pins on the pin layout should turn green.
- Set the system clock to 80 MHz.
- Select **PA2** and configure it as **USART2\_TX**. Also, select **PA3** and configure it as **USART2\_RX**.
- Expand the **Connectivity** tab and select **USART2**. Set its **Mode** to **Asynchronous**.
- In **Parameter Settings**, set the **Baud Rate** to 115200, **Word Length** to 8, **Parity** to None, and **Stop Bits** to 1.
- Configure pin **PA5** as **GPIO\_OUTPUT**.
- Configure **PC13** as **GPIO\_EXTI13**.
- Click **NVIC** under **System Core** and enable **EXTI line[15:10]** interrupts.

We now have to configure the X-CUBE-MEMS1 for our expansion board. The steps are given below.

- Click tab **Software Packs** and click **Select Components**.
- Expand tab **STMicroelectronics X-CUBE-MEMS1**.
- Enable **Board Extension** as **IKS01A2**.
- Expand **Device\_MEMS1 Application** and select IKS01A2\_Datalog as shown in Figure 16.18.

| Pack / Bundle / Component                       | Version | Selection                           |
|-------------------------------------------------|---------|-------------------------------------|
| Board Part Mag / LIS3MDL                        |         | Not selected                        |
| Board Part Mag / LIS2MDL                        |         | Not selected                        |
| Board Part Mag / IIS2MDC                        |         | Not selected                        |
| Board Part HumTemp / HTS221                     |         | Not selected                        |
| Board Part PressTemp / LPS22HB                  |         | Not selected                        |
| Board Part PressTemp / LPS22HH                  |         | Not selected                        |
| Board Part PressTemp / LPS33HW                  |         | Not selected                        |
| Board Part PressTemp / LPS33K                   |         | Not selected                        |
| Board Part Temp / STTS751                       |         | Not selected                        |
| Board Part Temp / STTS22H                       |         | Not selected                        |
| Board Part Gyr / A3G4250D                       |         | Not selected                        |
| Board Extension IKS01A3                         | 1.4.0   | <input type="checkbox"/>            |
| Board Extension IKS01A2                         | 5.3.1   | <input checked="" type="checkbox"/> |
| Board Extension IKS02A1                         | 1.1.0   | <input type="checkbox"/>            |
| Board Support STM32Cube_Custom_BSP_Driver:8.1.1 |         |                                     |
| Custom / MOTION_SENSOR                          |         | <input type="checkbox"/>            |
| Custom / ENV_SENSOR                             |         | <input type="checkbox"/>            |
| Device MEMS1_Applications                       | 8.1.1   |                                     |
| Application                                     |         | IKS01A2_DataLog...                  |
| Source STM32 MotionID Library                   | 2.2.1   |                                     |

Figure 16.18: Select Board Extension and Device\_MEMS1 Application.

Now we will configure the X-CUBE-MEMS1 library. The steps are as follows.

- Click to expand **Software Packs** at the left-hand side and click **STMicroelectronics X-CUBE-MEMS1** (Figure 16.19).


|                                                                                     |   |
|-------------------------------------------------------------------------------------|---|
| System Core                                                                         | > |
| Analog                                                                              | > |
| Timers                                                                              | > |
| Connectivity                                                                        | > |
| Multimedia                                                                          | > |
| Security                                                                            | > |
| Computing                                                                           | > |
| Middleware                                                                          | > |
| Software Packs                                                                      | > |
|  |   |
| ✓ STMicroelectronics X-CUBE-MEMS1.8...                                              |   |

Figure 16.19: Selecting Software Packs.

- Enable **Board Extension IKS01A2** and **Device MEMS1 Application**.
- Configure the Platform Settings as shown in Figure 16.20.

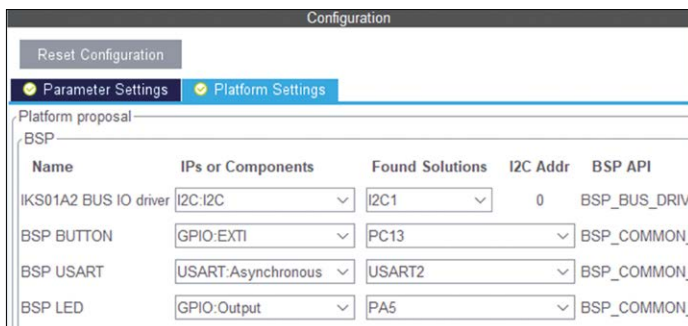


Figure 16.20: Configuring Platform Settings.

- Click **File**, followed by **Save** and click **YES** to generate code.
- Click **Core**, followed by **Src** and double click **main.c** to open the main program.

Compile the program (click **Project**, followed by **Build All**) in **Release** mode (Click **Project**, followed by **Build Configurations**, followed by **Set Active** and choose **Release**) making sure there are no errors. Drag and drop the binary file **MEMS2.bin** to device NUCLEO\_L476RG. Start the terminal emulator on your PC (e.g. SmartTrm, Putty, TeraTerm etc) and set the Baud rate to 115200, 8 data bits, no parity, 1 stop bit, no handshaking (i.e. no flow control). Figure 16.21 shows example output displayed on the screen. As you can see the following data are displayed every second: temperature (readings of LPS22HB and HTS221), humidity, pressure, gyroscope, accelerometer, and magnetometer.

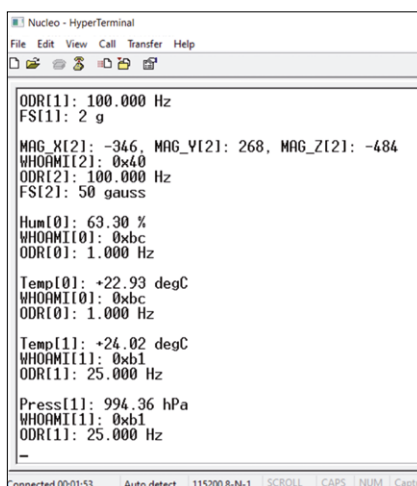


Figure 16.21: Example output on the screen.

The program listing without the comments, clock and I/O routines is shown in Figure 16.22. Notice that the program loop contains only the function: **MX\_MEMS\_Process()**.

```

/* USER CODE BEGIN Header */
/**

 * @file : main.c
 * @brief : Main program body

 * @attention
 *
 * <h2><center>© Copyright (c) 2020 STMicroelectronics.
 * All rights reserved.</center></h2>
 *
 * This software component is licensed by ST under BSD 3-Clause license,
 * the «License»; You may not use this file except in compliance with the
 * License. You may obtain a copy of the License at:
 *
 * opensource.org/licenses/BSD-3-Clause
 *

 */
#include «main.h»
#include «app_mems.h»

void SystemClock_Config(void);
static void MX_GPIO_Init(void);

int main(void)
{
 HAL_Init();

 SystemClock_Config();

 MX_GPIO_Init();
 MX_MEMS_Init();

 while (1)
 {
 MX_MEMS_Process();
 }
}

```

*Figure 16.22: Program: MEMS2.*

### 16.8 Wi-Fi Expansion Board (X-NUCLEO-IDW01M1)

This is the Wi-Fi expansion board based on the SPWF01SA module (Figure 16.23). The firmware supports up to eight TCP/UDP sockets as well as dynamic web pages with SSI, it can simultaneously behave as a socket server and socket client. TLS/SSL encryption is supported. WEP/WPA/WPA2 personal security is supported. The expansion board additionally includes 16 GPIOs. Interface with the host processor is via serial UART port. The AT

commands can be used to access various functions. The module is compatible with both Arduino and ST morpho connectors.

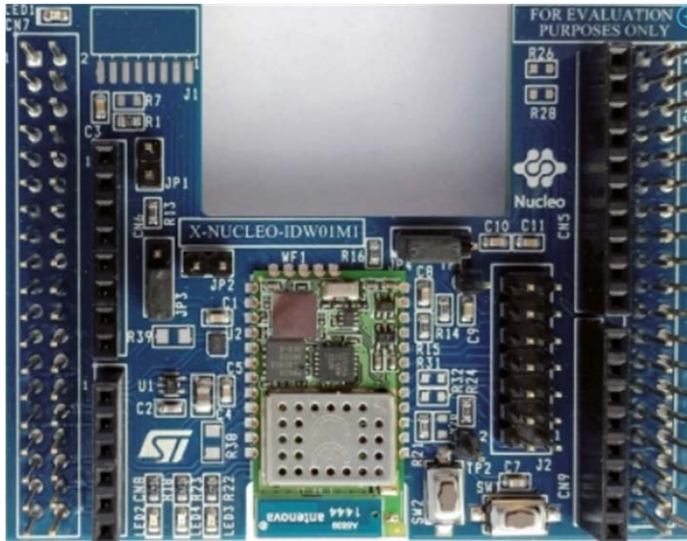


Figure 16.23: Wi-Fi expansion board.

Further information on this expansion board can be obtained from the following website:  
[http://www.st.com/content/st\\_com/en/products/ecosystems/stm32-open-development-environment/stm32-nucleo-expansion-boards/stm32-ode-connect-hw/x-nucleo-idw01m1.html](http://www.st.com/content/st_com/en/products/ecosystems/stm32-open-development-environment/stm32-nucleo-expansion-boards/stm32-ode-connect-hw/x-nucleo-idw01m1.html)

### User pushbuttons and LEDs

Two pushbuttons (SW1 and SW2) and four LEDs (LED 1, LED 2, LED 3, LED 4) are provided on the expansion board. The functions of these are as follows:

- SW1: Pushing this button resets the board
- SW2: This is the GPIO7\_ADC1 module signal
- LED 1: This green LED is lit to indicate 3.3V board power supply
- LED 2: This blue LED is at GPIO10
- LED 3: This red LED is at GPIO14
- LED 4: This yellow LED is at GPIO13

Figure 16.24 shows the functional block diagram of the Wi-Fi expansion board.

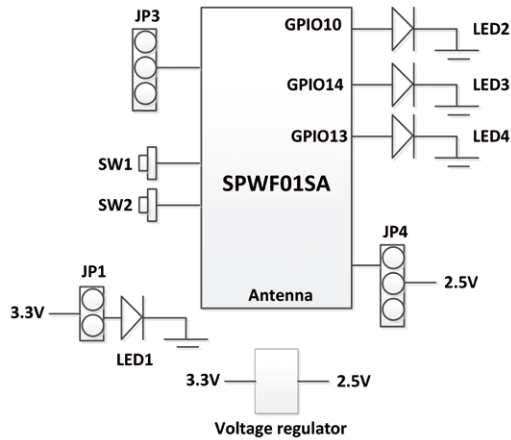


Figure 11.23: Functional block diagram.

### Program listing

A firmware called X-CUBE-WIFI1 is available for the X-NUCLEO-IDW01M1 expansion board at the following web site (go to the end of the page and click **Get Software** to download the software):

[http://www.st.com/content/st\\_com/en/products/embedded-software/mcus-embedded-software/stm32-embedded-software/stm32cube-expansion-packages/x-cube-wifi1.html](http://www.st.com/content/st_com/en/products/embedded-software/mcus-embedded-software/stm32-embedded-software/stm32cube-expansion-packages/x-cube-wifi1.html)

The firmware contains all the project codes for an application, developed for different types of Nucleo boards. Interested readers can copy and unzip this firmware to their computers.

## APPENDIX • FreeRTOS For the STM32 MCU

### A.1 Overview

Most complex real-time systems require a number of tasks (or programs) to be processed almost at the same time. For example, consider an extremely simple real-time system which is required to flash an LED at required intervals, and at the same time monitor for a key input from a keyboard. One solution would be to scan the keyboard in a loop at regular intervals while flashing the LED at the same time. Although this approach may work for a simple example, in most complex real-time systems a multitasking approach should be implemented.

The term multitasking means that several tasks (or programs) are processed in parallel on the same CPU. However, it is not possible for several tasks to run on a single CPU at the same time. Therefore, task switching is done where the tasks share the CPU time. In many applications, tasks cannot run independently of each other and they are expected to co-operate in some way. For example, the execution of a task may depend upon the completion of another, or a task may need some data from another. In such circumstances the tasks involved must be synchronized using some form of inter-task communication method.

Real-time systems are time responsive systems where the CPU is never overloaded. In such systems, tasks usually have priorities which are obeyed strictly. A task with a higher priority can grab the CPU from a lower-priority task and then use the CPU exclusively until it releases the CPU. When the higher-priority task completes its processing, or if it is waiting for a resource to become available, then the lower priority task can grab the CPU and resume processing from the point where it was interrupted. Real-time systems are also expected to react to events as quickly as possible. External events are usually processed using external interrupts and the interrupt latency of such systems is expected to be very short so that the interrupt service routine is executed as soon as an interrupt occurs.

### A.2 Multitasking kernel advantages

- Without a multitasking kernel, multiple tasks can be executed in a loop, but this approach results in very poorly controlled real-time performance where the execution times of the tasks cannot be controlled.
- It is possible to code the various tasks as interrupt service routines. This may work in practice, but if the application has many tasks then the number of interrupts grow, making the code less manageable.
- A multitasking kernel allows new tasks to be added or some of the existing tasks to be removed from the system without any difficulty.
- The testing and debugging of a multitasking system with a multitasking kernel is easy compared to a multitasking system without a kernel.
- Memory is better managed using a multitasking kernel.
- Inter-task communication is easily handled using a multitasking kernel.
- Task synchronization is easily controlled using a multitasking kernel.
- CPU time is easily managed using a multitasking kernel.
- Most multitasking kernels provide memory security where a task cannot access the memory space of another task.

- Most multitasking kernels provide task priority where higher priority tasks can grab the CPU and stop the execution of lower priority tasks. This allows important tasks to run whenever it is required.

### A.3 The need for an RTOS

An RTOS (Real-Time Operating System) is a program that manages system resources, schedules the execution of various tasks in a system, synchronizes the execution of tasks, manages resource allocations, and provides inter-task communication and messaging between the tasks. Every RTOS consists of a kernel that provides the low-level functions, mainly the scheduling, task creation, inter-task communication, resource management etc. Most complex RTOSs also provide file-handling services, disk read-write operations, interrupt servicing, network management, user management, etc.

A task is an independent thread of execution in a multitasking system, usually with its own local set of data. A multitasking system consists of several tasks, each running its own code, and communicating and synchronizing with each other in order to have access to shared resources. Perhaps the easiest example of a multitasking system is where there are say 3 LEDs and each LED flashes at a different rate. The programming of such a simple system without a multitasking kernel could be a complicated task. We will see in this Appendix how a multitasking operating system such as the FreeRTOS can be used to share the MCU resources.

The simplest RTOS consists of a scheduler that determines the execution order of the tasks in the system. Each task has its own context consisting of the state of the CPU and its associated registers. The scheduler switches from one task to another one by performing a context switching where the context of the running task is stored and the context of the next task is loaded appropriately so that execution can continue properly with the next task. The time taken for the CPU to perform context switching is known as the context switching time and is usually negligible compared to the actual execution times of the tasks.

### A.4 The FreeRTOS

FreeRTOS is a real-time operating system that runs on many high-end microcontrollers, including the STM32 family. STM32CubeIDE has bundled FreeRTOS software, although we will not be making calls to FreeRTOS directly. ARM has created the CMSIS-RTOS library, which allows us to make calls to an underlying RTOS such as the FreeRTOS, i.e. we will be making calls to CMSIS-RTOS (version 2) in order to control the underlying FreeRTOS.

FreeRTOS and multitasking is a very large topic and requires several books to explain all of its features. There are several books, tutorials and application notes on the internet that may be helpful for the readers who are new to the concepts of multitasking. The following book can be very helpful to understand the concepts of multitasking and learn to use the FreeRTOS in ARM based MCUs:

**Book:** *ARM-Based-Microcontroller-Multitasking-Projects* — Using the FreeRTOS

**Author:** Dogan Ibrahim

**Amazon Link:** [https://www.amazon.co.uk/ARM-Based-Microcontroller-Multitasking-ProjectsFreeRTOS/dp/0128212276/ref=sr\\_1\\_1?dchild=1&keywords=dogan+ibrahim&qid=1602082838&sr=8-1](https://www.amazon.co.uk/ARM-Based-Microcontroller-Multitasking-ProjectsFreeRTOS/dp/0128212276/ref=sr_1_1?dchild=1&keywords=dogan+ibrahim&qid=1602082838&sr=8-1)



### A.5 FreeRTOS project with the STM32CubeIDE

In the remainder of this Appendix we will develop a simple application with 3 LEDs connected to the NUCLEO-L476RG development board. The LEDs are named as **LEDSLOW**, **LEDMEDIUM**, and **LEDFAST**. The connections of these LEDs and their flashing rates are as follows (Figure A.1):

| LED       | Flash rate   | Connected to GPIO pin |
|-----------|--------------|-----------------------|
| LEDSLOW   | every second | PC0                   |
| LEDMEDIUM | every 500 ms | PC1                   |
| LEDFAST   | every 250 ms | PC2                   |

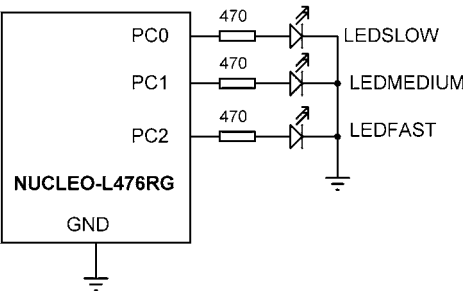


Figure A.1: Circuit diagram of the example.

#### Program listing

The steps are as follows.

- Start STM32CubeIDE.
- Select to start a new application.
- Create a new workspace.
- Select STM32L476RG as the processor.
- Name the program as **FREE**.
- Configure **PC0**, **PC1**, and **PC2** as **GPIO\_Output**. Right click on the pins and set the **User Labels** to **LEDSLOW**, **LEDMEDIUM**, and **LEDFAST** respectively (Figure A.2)

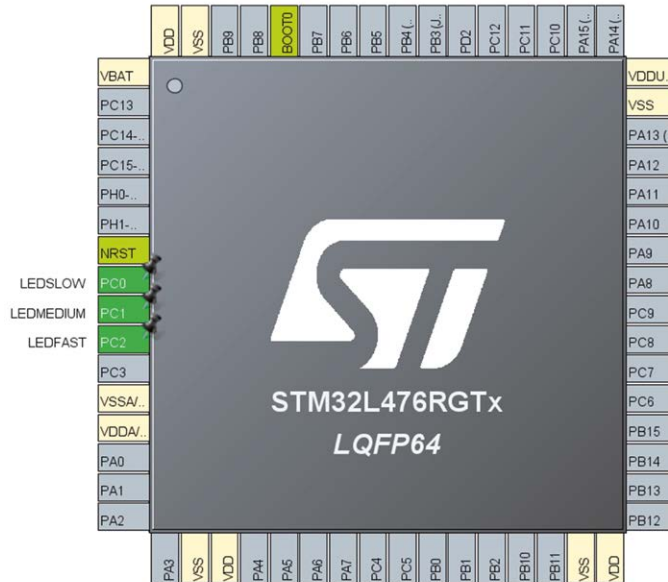


Figure A.2: Configuring the outputs.

- Configure the MCU clock as 80 MHz.
- Click **Middleware** at the left hand side and select **FreeRTOS**.
- Set the **Interface** to **CMSIS\_V2**.
- There are many parameters that can be configured under the Configuration tab. The use of these parameters requires good knowledge of the FreeRTOS. In this project we will accept all the default values (see Figure A.3).

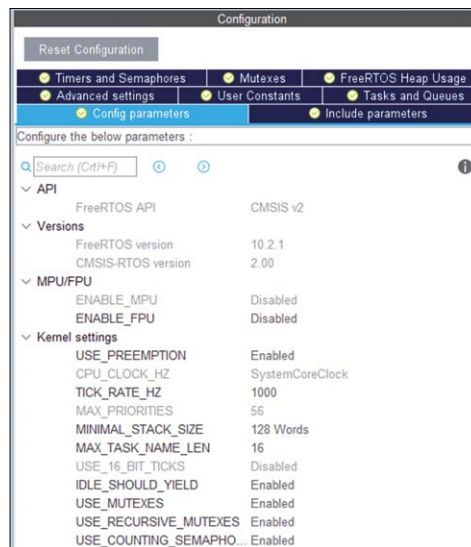


Figure A.3: Accept the default values.

- Click **File**, followed by **Save** and click **YES** to generate code.
- Click **Core**, followed by **Src** and double click **main.c** to display the main program.

In this program we will have 3 tasks, also called threads. The basic functions in a FreeRTOS program are as follows:

- define thread IDs;
- define thread attributes;
- initialize the scheduler;
- create threads;
- start the scheduler.

**osThreadId\_t** is used to define the thread IDs. The thread IDs in this program are:

```
osThreadId_t LEDSTaskHandle; // Slow LED
osThreadId_t LEDMTaskHandle; // Medium LED
osThreadId_t LEDFTaskHandle; // Fast LED
```

The thread attributes define various parameters of a thread, such as its name, priority, stack size etc. As an example, the thread attribute for the slow task are:

```
const osThreadAttr_t LEDSTask_attributes =
{
 .name = "LEDSTask",
 .priority = (osPriority_t) osPriorityNormal,
 .stack_size = 128 * 4
};
```

The scheduler is initialized and started with the function calls:

```
osKernelInitialize();
osKernelStart();
```

Threads are created by using function call **osThreadNew()**. For example, the thread for the slow LED is created as follows:

```
LEDSTaskHandle = osThreadNew(StartLEDSTask, NULL, &LEDSTask_attributes);
```

Where, **StartLEDSTask** is the name of the function that will be called by the scheduler to implement the slow flashing LED. Its contents are:

```
void StartLEDSTask(void *argument)
{
 for(;;)
 {
 HAL_GPIO_TogglePin(GPIOC, LEDSLow_Pin);
 }
}
```

```

 osDelay(1000);
 }
}

```

Note that instead of **HAL\_Delay()**, we need to use **osDelay()**. **HAL\_Delay()** in a high priority task might prevent a context switch. But the **osDelay()** tells the scheduler to switch to a different task while waiting.

Compile the program in **Release** mode and drag and drop the binary file **FREE.bin** to device NUCLEO-L476RG. At this point all 3 threads will be running simultaneously in their own forever loops. The scheduler will switch the threads in and out to give the appearance that we are running three threads at the same time. We should see the three LEDs flashing at different rates simultaneously.

Figure A.4 shows the program listing (Program: FREE), where the comments have been removed for clarity.

```

/* USER CODE BEGIN Header */
/**

 * @file : main.c
 * @brief : Main program body

 * @attention
 *
 * <h2><center>© Copyright (c) 2020 STMicroelectronics.
 * All rights reserved.</center></h2>
 *
 * This software component is licensed by ST under Ultimate Liberty license
 * SLA0044, the «License»; You may not use this file except in compliance with
 * the License. You may obtain a copy of the License at:
 *
 * www.st.com/SLA0044
 *

 */
#include «main.h»
#include «cmsis_os.h»

//
// Define Thread IDs
//
osThreadId_t LEDSTaskHandle; // Slow LED
osThreadId_t LEDMTaskHandle; // Medium LED
osThreadId_t LEDFTaskHandle; // Fast LED

//
// Slow LED task. Flash every second

```

```
//
void StartLEDSTask(void *argument)
{
 for(;;)
 {
 HAL_GPIO_TogglePin(GPIOC, LEDSLOW_Pin);
 osDelay(1000);
 }
}

//
// Medium LED task. Flash every 500ms
//
void StartLEDMTask(void *argument)
{
 for(;;)
 {
 HAL_GPIO_TogglePin(GPIOC, LEDMEDIUM_Pin);
 osDelay(500);
 }
}

//
// Fast LED task. Flash every 250ms
//
void StartLEDFTask(void *argument)
{
 for(;;)
 {
 HAL_GPIO_TogglePin(GPIOC, LEDFAST_Pin);
 osDelay(250);
 }
}

void SystemClock_Config(void);
static void MX_GPIO_Init(void);
void StartDefaultTask(void *argument);

//
// Start of main program
//
int main(void)
{
 HAL_Init();
 SystemClock_Config();
 MX_GPIO_Init();
```

```

//
// Slow LED Task attributes
//
const osThreadAttr_t LEDSTask_attributes =
{
 .name = «LEDSTask»,
 .priority = (osPriority_t) osPriorityNormal,
 .stack_size = 128 * 4
};

//
// Medium LED Task attributes
//
const osThreadAttr_t LEDMTask_attributes =
{
 .name = «LEDMTask»,
 .priority = (osPriority_t) osPriorityNormal,
 .stack_size = 128 * 4
};

//
// Fast LED Task attributes
//
const osThreadAttr_t LEDFTask_attributes =
{
 .name = «LEDFTask»,
 .priority = (osPriority_t) osPriorityNormal,
 .stack_size = 128 * 4
};

/* Init scheduler */
osKernelInitialize();

/* creation of Tasks */
LEDSTaskHandle = osThreadNew(StartLEDSTask, NULL, &LEDSTask_attributes);
LEDMTaskHandle = osThreadNew(StartLEDMTask, NULL, &LEDMTask_attributes);
LEDFTaskHandle = osThreadNew(StartLEDFTask, NULL, &LEDFTask_attributes);

/* Start scheduler */
osKernelStart();

while (1)
{
}
}

```

```
void SystemClock_Config(void)
{
 RCC_OscInitTypeDef RCC_OscInitStruct = {0};
 RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

 RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
 RCC_OscInitStruct.HSISState = RCC_HSI_ON;
 RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
 RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
 RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
 RCC_OscInitStruct.PLL.PLLM = 2;
 RCC_OscInitStruct.PLL.PLLN = 20;
 RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV7;
 RCC_OscInitStruct.PLL.PLLQ = RCC_PLLQ_DIV2;
 RCC_OscInitStruct.PLL.PLLR = RCC_PLLR_DIV2;
 if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
 {
 Error_Handler();
 }

 RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
 |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
 RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
 RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
 RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
 RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

 if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_4) != HAL_OK)
 {
 Error_Handler();
 }

 if (HAL_PWREx_ControlVoltageScaling(PWR_REGULATOR_VOLTAGE_SCALE1) != HAL_OK)
 {
 Error_Handler();
 }
}

static void MX_GPIO_Init(void)
{
 GPIO_InitTypeDef GPIO_InitStruct = {0};

 /* GPIO Ports Clock Enable */
 __HAL_RCC_GPIOC_CLK_ENABLE();
```

```
/*Configure GPIO pin Output Level */
HAL_GPIO_WritePin(GPIOC, LEDSLow_Pin|LEDMEDIUM_Pin|LEDFAST_Pin, GPIO_PIN_RESET);

/*Configure GPIO pins : LEDSLow_Pin LEDMEDIUM_Pin LEDFAST_Pin */
GPIO_InitStruct.Pin = LEDSLow_Pin|LEDMEDIUM_Pin|LEDFAST_Pin;
GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
GPIO_InitStruct.Pull = GPIO_NOPULL;
GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);
}

void Error_Handler(void)
{

}

#ifdef USE_FULL_ASSERT

void assert_failed(uint8_t *file, uint32_t line)
{

}

#endif

/***** (C) COPYRIGHT STMicroelectronics *****/
```

*Figure A.4: FREE program listing.*



## Index

### Symbols

|                                |          |                                |          |
|--------------------------------|----------|--------------------------------|----------|
| 7-Segment LED                  | 132      | CMSIS-RTOS                     | 485      |
| <b>A</b>                       |          | CN1                            | 21       |
| ACK                            | 384      | CN2                            | 22       |
| ADC                            | 48       | CN5                            | 23       |
| ADC calibration                | 276      | CN6                            | 21, 23   |
| ADC clock cycles               | 227      | CN7                            | 21, 23   |
| ADC conversion rate            | 228      | CN8                            | 23       |
| ADC DMA                        | 275      | CN9                            | 23       |
| ADC in polling mode            | 246      | CN10                           | 22, 23   |
| ADC with DMA                   | 266      | common-anode                   | 133      |
| AHB prescaler                  | 48       | common-cathode                 | 133      |
| AHB Prescaler                  | 79       | Configure trace                | 435      |
| analogue input                 | 325      | Continuous Conversion Mode     | 239, 363 |
| analogue temperature sensor    | 251      | contrast control pin           | 206      |
| Analogue to Digital Converters | 226      | Conversion Interrupt           | 246      |
| analogue watchdog              | 228      | CoreMark                       | 29       |
| APB1 prescaler                 | 48       | Cortex-A                       | 29       |
| APB1 Prescaler                 | 79       | Cortex architecture            | 12       |
| APB2 prescaler                 | 48       | Cortex-M                       | 28       |
| APB2 Prescaler                 | 79       | Cortex-R                       | 28       |
| Arbitrary Waveform             | 287      | Counter Mode                   | 191      |
| ARDUINO                        | 370      | Crystal oscillator             | 47       |
| Arduino Uno R3 connector       | 453      | CPHA                           | 416      |
| Asynchronous                   | 340      | CPOL                           | 416      |
| auto-reload preload            | 191, 309 | CTRL_REG1                      | 466      |
| auto-reload register           | 306      | CTS                            | 336      |
| Auto reload register           | 191      | current limiting resistors     | 135      |
| <b>B</b>                       |          | <b>D</b>                       |          |
| Baud rate                      | 336      | DAC                            | 48       |
| Board Extension                | 479      | data overrun                   | 228      |
| Build All                      | 92       | DC56-11EWA                     | 141      |
| Byte Writing                   | 397      | DC motor control               | 459      |
| <b>C</b>                       |          | Debugging                      | 428      |
| calibration factor             | 276      | demo software                  | 24       |
| Callback function              | 179      | Device_MEMS1 Application       | 479      |
| C/C++Build                     | 430      | device opcode                  | 386      |
| Circular mode                  | 275      | Digital-to-Analogue Converters | 277      |
| clock circuit                  | 45       | direct-memory-access           | 187      |
| Clock Configuration            | 86       | Discontinuous conversion mode  | 226      |
| clock source                   | 187      | DMA                            | 267      |
|                                |          | DMA based ADC                  | 266      |
|                                |          | DMA Continuous Requests        | 267      |
|                                |          | DMA mode                       | 228, 382 |

**E**

|                                |          |
|--------------------------------|----------|
| E5V                            | 18, 21   |
| end of conversion              | 228      |
| End of Conversion Selection    | 230, 268 |
| end of injected conversion     | 228      |
| End of sequence of conversion  | 268      |
| End of sequence of conversions | 230      |
| EEPROM                         | 395      |
| EWARM                          | 56       |
| Expansion Boards               | 453      |
| Export                         | 88       |
| external crystal               | 86       |
| External-driven ADC            | 276      |
| External interrupt             | 148      |
| External interrupts            | 50, 51   |
| EXTI                           | 51       |
| EXTI_FTSR                      | 54       |
| EXTI_IMR                       | 54       |
| EXTI line0 interrupt           | 172      |
| EXTI line1 interrupt           | 172      |
| EXTI_PR                        | 55       |
| EXTI registers                 | 54       |
| EXTI_RTSR                      | 54       |

**F**

|              |     |
|--------------|-----|
| FIFO_CTRL    | 466 |
| flash memory | 14  |
| Flow control | 336 |
| FreeRTOS     | 484 |

**G**

|                       |        |
|-----------------------|--------|
| General-purpose timer | 189    |
| GPIO                  | 34     |
| GPIO library          | 72     |
| GPIOx -> IDR          | 74     |
| GPIOx -> ODR          | 74     |
| GPIOx_AFRH            | 39, 41 |
| GPIOx_AFRL            | 39     |
| GPIOx_BSRR            | 38     |
| GPIOx_IDR             | 36     |
| GPIOx_LCKR            | 38     |
| GPIOx_MODER           | 39     |
| GPIOx_ODR             | 36     |
| GPIOx_OSPEEDR         | 39, 40 |
| GPIOx_OTYPER          | 39     |
| GPIOx_PUPDR           | 39, 41 |

**H**

|                               |        |
|-------------------------------|--------|
| HAL_ADC_ConvCpltCallback      | 276    |
| HAL_ADC_ConvHalfCpltCallback  | 276    |
| HAL_ADC_Start_IT              | 247    |
| HAL_DAC_Start_DMA             | 289    |
| half-duplex                   | 383    |
| HAL_GPIO_EXTI_Callback        | 74     |
| HAL_GPIO_EXTI_IRQHandler      | 74     |
| HAL_GPIO_LockPin              | 74     |
| HAL_GPIO_ReadPin              | 73     |
| HAL_GPIO_TogglePin            | 74     |
| HAL_GPIO_WritePin             | 73     |
| HAL_Init                      | 72     |
| HAL_TIM_PeriodElapsedCallback | 280    |
| HAL_UART_Receive_IT           | 376    |
| HAL_UART_Transmit_IT          | 376    |
| HCLK                          | 48     |
| Hitachi HD44780               | 205    |
| HSE                           | 46     |
| HSI                           | 47, 79 |
| HTS221                        | 461    |
| HyperTrm                      | 337    |

**I**

|                           |     |
|---------------------------|-----|
| I2C bus                   | 383 |
| I2C Speed Frequency       | 389 |
| I2C Speed Mode            | 389 |
| IDD                       | 22  |
| IKS01A2                   | 479 |
| ingle-ended               | 363 |
| Internal clock            | 191 |
| Internal Clock Division   | 191 |
| interrupt mode            | 228 |
| interrupt priority        | 182 |
| Interrupts                | 49  |
| Interrupt Service Routine | 49  |
| Interrupt sources         | 50  |
| Inter-task communication  | 484 |
| ISO8200BQ                 | 453 |
| IWDG                      | 443 |

**J**

|            |    |
|------------|----|
| JP5        | 21 |
| jumper JP1 | 25 |
| Jumper JP6 | 22 |

## L

|                      |        |
|----------------------|--------|
| LCD                  | 205    |
| lcd_clear            | 209    |
| lcd_goto             | 209    |
| lcd_init             | 209    |
| lcd_putch            | 209    |
| lcd_puts             | 209    |
| LCD_STROBE           | 209    |
| lcd_write_cmd        | 209    |
| lcd_write_data       | 209    |
| LD1                  | 22, 25 |
| LD2                  | 22, 25 |
| LM35                 | 226    |
| Low-power run mode   | 441    |
| Low-power sleep mode | 442    |
| LPS22HB              | 461    |
| LPTIM                | 445    |
| LPUART               | 445    |
| LSCO                 | 48     |
| LSE                  | 47     |
| LSM6DSL              | 461    |
| LSM303AGR            | 461    |

## M

|                            |        |
|----------------------------|--------|
| Maximum Triangle Amplitude | 298    |
| Mbed                       | 58, 67 |
| Mbed Compiler              | 59     |
| MCO                        | 47     |
| MCP23S17                   | 417    |
| MCU GCC Compiler           | 430    |
| MCU GCC Linker             | 437    |
| MDK-ARM                    | 60     |
| memory security            | 484    |
| MAX232                     | 335    |
| MCP23017                   | 385    |
| MISO                       | 415    |
| Morpho connector           | 453    |
| MOSI                       | 415    |
| MSI RC                     | 47     |
| Multiple Inputs            | 237    |
| Multiple PWM               | 318    |
| multiple slave             | 383    |
| multiplexed 7-segment LED  | 195    |
| multiplexed LED            | 140    |
| multitasking kernel        | 148    |
| MX_GPIO_Init               | 73     |

## N

|                 |         |
|-----------------|---------|
| Noise Amplitude | 304     |
| Noise signal    | 303     |
| Noise waveform  | 304     |
| NPN transistors | 164     |
| Nucleo-32       | 15      |
| Nucleo-64       | 16      |
| Nucleo-144      | 17      |
| NVIC            | 50, 151 |

## O

|                            |          |
|----------------------------|----------|
| ON-OFF temperature control | 258      |
| open-drain                 | 34       |
| Open-drain                 | 35       |
| operating temperature      | 15       |
| Optimization               | 429      |
| osDelay                    | 489      |
| osThreadId_t               | 488      |
| osThreadNew                | 488      |
| Overrun behaviour          | 230, 363 |
| Overrun data overwritten   | 230, 363 |
| overrun flag               | 228      |

## P

|                     |     |
|---------------------|-----|
| Page Writing        | 397 |
| Parallel LCDs       | 205 |
| Parity              | 336 |
| Part Number Search  | 77  |
| PDL                 | 124 |
| peak amplitude      | 297 |
| PLL                 | 47  |
| PLLM                | 79  |
| PLL multiplier      | 79  |
| PLLSAI1             | 227 |
| PLLSAI2             | 227 |
| PLL Source Mux      | 79  |
| polling mode        | 228 |
| pow                 | 106 |
| Power Management    | 439 |
| power supply        | 21  |
| Preemptio priority  | 151 |
| Preemptive priority | 156 |
| prescaler           | 189 |
| Prescaler           | 191 |
| pressure sensor     | 461 |
| PRIMASK             | 50  |

|                                    |     |                                   |          |
|------------------------------------|-----|-----------------------------------|----------|
| printf                             | 436 | Serial Wire                       | 429      |
| pull-down                          | 34  | Serial Wire Viewer                | 434      |
| pull-up                            | 34  | SFRs                              | 433      |
| pulse-width-modulated              | 187 | Shutdown mode                     | 444      |
| Pulsewidth Modulation              | 305 | Sinewave                          | 286      |
| pushbutton switches                | 22  | Single continuous conversion mode | 226      |
| push-pull                          | 34  | Single conversion mode            | 226      |
| Putty                              | 337 | single-ended                      | 383      |
| PWM waveform                       | 305 | Single-ended                      | 267, 363 |
| PWR_FLAG                           | 451 | single master                     | 383      |
|                                    |     | sleep mode                        | 439      |
|                                    |     | Sleep mode                        | 441      |
| <b>R</b>                           |     | SoftwareSerial library            | 370      |
| rand                               | 106 | SPI Bus                           | 415      |
| RandomFlash                        | 106 | sprintf                           | 253      |
| Rank 1 Channel                     | 239 | SPWF01SA                          | 481      |
| Rank 2 Channel                     | 239 | Squarewave                        | 285      |
| Read/Write                         | 206 | SRAM2                             | 443      |
| Real-time systems                  | 484 | srand                             | 126      |
| Register Select                    | 206 | SSEL                              | 415      |
| Release mode                       | 92  | Standby mode                      | 443      |
| Resume                             | 431 | STATUS                            | 467      |
| RS232 standard                     | 334 | stdlib.h                          | 125      |
| RTC                                | 43  | Step into                         | 431      |
| RTC alarm                          | 443 | Step over                         | 431      |
| RTC_FLAG                           | 451 | ST-LINK                           | 12, 18   |
| RTOS                               | 485 | ST-LINKV2-1                       | 22       |
| RTS                                | 336 | stm32l4xx_it.c                    | 179      |
| Run mode                           | 439 | STM32L476RGT6                     | 31       |
| Run to line                        | 431 | STM32 L family                    | 12       |
|                                    |     | ST Morpho                         | 12       |
| <b>S</b>                           |     | Stop 1 mode                       | 443      |
| Sample & Hold capacitor            | 227 | Stop 2 mode                       | 443      |
| Saving the program                 | 88  | Stop bits                         | 336      |
| Sawtooth Waveform                  | 277 | string.h                          | 341      |
| Scan Conversion Mode               | 239 | STSPIN250                         | 459      |
| Scan multi-channel continuous      |     | Subpriority                       | 151      |
| conversion mode                    | 226 | Subpriority priority              | 156      |
| Scan multi-channel conversion mode | 226 | SW4STM32                          | 62       |
| SCK                                | 415 | SWCLK                             | 18, 428  |
| SCL                                | 383 | SWDIO                             | 18, 428  |
| SDA                                | 383 | SWV ITM Data Console              | 435      |
| SDI                                | 415 | SYSCFG_EXTICRx                    | 52       |
| SDO                                | 415 | SYSClk                            | 45       |
| seed value                         | 126 | System Core                       | 150      |
| Serial communication               | 334 | System Workbench                  | 62       |
| serial LCDs                        | 205 |                                   |          |

|                              |          |              |     |
|------------------------------|----------|--------------|-----|
| <b>T</b>                     |          | Word length  | 336 |
| Target Selection             | 77       |              |     |
| Task synchronization         | 484      | <b>X</b>     |     |
| TCP/UDP                      | 481      | X-CUBE-MEMS1 | 468 |
| temperature sensor           | 226      |              |     |
| TEMP_OUT_H                   | 464      |              |     |
| TEMP_OUT_L                   | 464      |              |     |
| TeraTerm                     | 337      |              |     |
| Terminate                    | 431      |              |     |
| Terminate and Relaunch       | 431      |              |     |
| Thumb 2                      | 28       |              |     |
| time.h                       | 125      |              |     |
| timer-based DMA              | 289      |              |     |
| Timer-driven ADC             | 276      |              |     |
| timer interrupt              | 148      |              |     |
| timer interrupts             | 190      |              |     |
| Timers                       | 49       |              |     |
| TMP36                        | 252      |              |     |
| TMP102                       | 405      |              |     |
| Tool Settings                | 429      |              |     |
| TRGO                         | 276      |              |     |
| Triangle wave generation     | 298      |              |     |
| triangular waves             | 296      |              |     |
| Trigger Event Selection TRGO | 290      |              |     |
| TrueSTUDIO                   | 61       |              |     |
| TTL level signals            | 334      |              |     |
| <b>U</b>                     |          |              |     |
| UART ports                   | 336      |              |     |
| Up Counter                   | 221, 345 |              |     |
| Up/Down Counter              | 170      |              |     |
| USB connector                | 21       |              |     |
| <b>V</b>                     |          |              |     |
| Variable Duty Cycle          | 316      |              |     |
| VBAT                         | 43       |              |     |
| VDDUSB                       | 43       |              |     |
| VEE                          | 206      |              |     |
| <b>W</b>                     |          |              |     |
| Wave generation mode         | 298, 304 |              |     |
| WFE                          | 444      |              |     |
| WFI                          | 444      |              |     |
| Wi-Fi expansion board        | 481      |              |     |
| WKUP 1                       | 451      |              |     |
| WKUP pin                     | 444      |              |     |

# Nucleo Boards Programming with the STM32CubeIDE

Hands-on in more than 50 projects

STM32 Nucleo family of processors are manufactured by STMicroelectronics. These are low-cost ARM microcontroller development boards. This book is about developing projects using the popular STM32CubeIDE software with the Nucleo-L476RG development board. In the early Chapters of the book the architecture of the Nucleo family is briefly described.

The book covers many projects using most features of the Nucleo-L476RG development board where the full software listings for the STM32CubeIDE are given for each project together with extensive descriptions. The projects range from simple flashing LEDs to more complex projects using modules, devices, and libraries such as GPIO, ADC, DAC, I2C, SPI, LCD, DMA, analogue inputs, power management, X-CUBE-MEMS1 library, DEBUGGING, and others. In addition, several projects are given using the popular Nucleo Expansion Boards. These Expansion Boards plug on top of the Nucleo development boards and provide sensors, relays, accelerometers, gyroscopes, Wi-Fi, and many others. Using an expansion board together with the X-CUBE-MEMS1 library simplifies the task of project development considerably.

All the projects in the book have been tested and are working. The following sub-headings are given for each project: Project Title, Description, Aim, Block Diagram, Circuit Diagram, and Program Listing for the STM32CubeIDE.

## In this book you will learn about

- STM32 microcontroller architecture;
- the Nucleo-L476RG development board in projects using the STM32CubeIDE integrated software development tool;
- external and internal interrupts and DMA;
- DEBUG, a program developed using the STM32CubeIDE;
- the MCU in Sleep, Stop, and in Standby modes;
- Nucleo Expansion Boards with the Nucleo development boards.

## What you need

- a PC with Internet connection and a USB port;
- STM32CubeIDE software (available at STMicroelectronics website free of charge)
- the project source files, available from the book's webpage hosted by Elektor;
- Nucleo-L476RG development board;
- simple electronic devices such as LEDs, temperature sensor, I2C and SPI chips, and a few more;
- Nucleo Expansion Boards (optional).



**Prof. Dr. Dogan Ibrahim** has BSc degree in electronic engineering, an MSc degree in automatic control engineering, and a PhD degree in digital signal processing. Dogan has worked in many industrial organizations before he returned to academic life. He is the author of over 60 technical books and over 200 technical articles on microcontrollers, microprocessors, and related fields. He is a Chartered electrical engineer and a Fellow of the Institution of Engineering Technology.

**Elektor International Media BV**  
www.elektor.com

